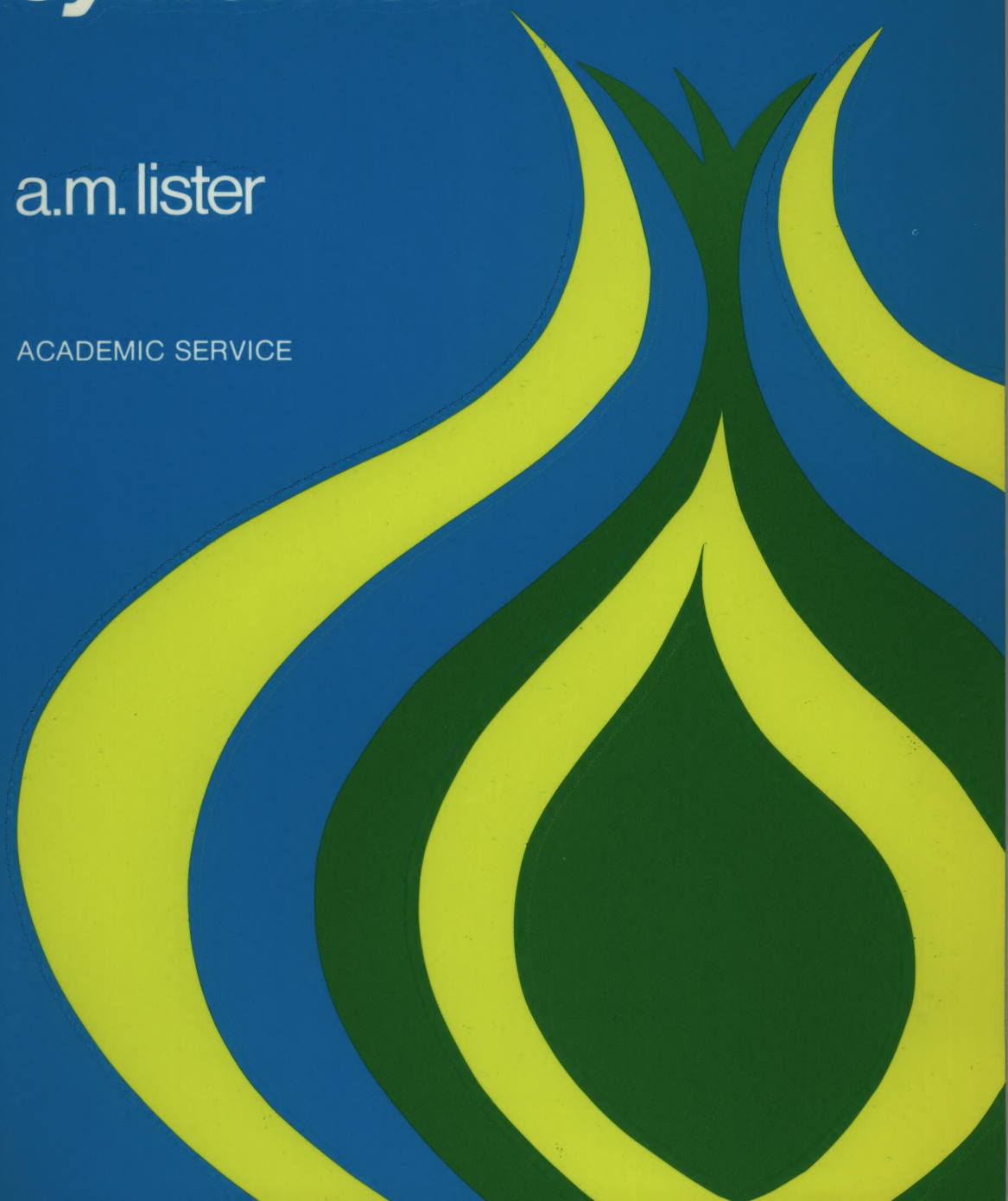
inleiding besturings-systemen

a.m. lister

ACADEMIC SERVICE



INLEIDING BESTURINGSSYSTEMEN

Andere boeken op het gebied van besturingssystemen:

Bedrijfssystemen - *EIT-4*

Systeemprogrammatuur en software-ontwikkeling voor micro-computers - *Verhulst*

CP/M - *Fernandez/Ashley*

CP/M systeemkaart

CP/M-86 - *Fernandez/Ashley*

CP/M voor gevorderden - *Clarke e.a.*

PC DOS - *Ashley/Fernandez*

MS DOS - *Ashley/Fernandez*

UNIX - *Austen/Thomassen*

Systeemprogrammatuur, een inleiding met de machinestructuur van de PDP11 als voorbeeld - *Alblas*

Werken met UNIX - *Kernighan/Pike* (in voorbereiding)

inleiding besturings-systemen

a.m. lister

ACADEMIC SERVICE

Published in the United Kingdom by The Macmillan Press under the title *Fundamentals of Operating Systems 3rd ed.*

Nederlandse vertaling:
Directie: Amstelveens vertaalburo, vertaler Paul Paulssen

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Lister, A.M.

Inleiding besturingssystemen / A.M. Lister ; [vert. uit het Engels door P. Paulssen]. - Den Haag : Academic Service. - Ill.
Vert. van: Fundamentals of operating systems. - 3e dr. - Londen [etc.] : Macmillan, 1984. - (Macmillan computer science series).
- Oorspr. uitg.: 1979. - Met lit. opg.
ISBN 90 6233 165 3
SISO 365.3 SVS 8.12.3 UDC 681.3.066 UGI 200
Trefw.: operating systems.

Uitgegeven door: Academic Service
Postbus 96996
2509 JJ Den Haag
Zetwerk: INFOTYPE, Maarheeze
Omslagontwerp: JAM Gauw
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam
ISBN 90 6233 165 3

aan mijn ouders

Inhoud

Voorwoord bij de eerste editie

Een besturingssysteem is waarschijnlijk het belangrijkste deel van de verzameling software die bij iedere moderne computer hoort. Het belang wordt weergegeven door de hoeveelheid uren die gewoonlijk in de ontwikkeling ervan gestoken worden en door de mystiek waarmee het vaak omhuld wordt. Voor een leek is het ontwerpen en het bouwen van besturingssystemen vaak een ontoegankelijke aangelegenheid, althans voor buitenstaanders. Ik hoop dat dit boek op de een of andere manier bijdraagt tot het verdrijven van die mystiek en tot een beter begrip van de principes aan de hand waarvan besturingssystemen ontworpen worden.

Het materiaal is gebaseerd op een reeks lezingen die ik de afgelopen paar jaar gegeven heb aan vóórkandidaats-studenten in de informatica. Het boek is daarom een geschikte inleiding voor studenten die wat basiskennis hebben van informatica, of voor mensen die enige tijd met computers gewerkt hebben. In het ideale geval zou de lezer enige kennis moeten hebben van programmeren en zou hij bekend moeten zijn met algemene machine-architectuur, gewone gegevensstructuren zoals lijsten en bomen, en de functies van systeemsoftware zoals loaders en editors. Het zou evenzeer geschikt zijn als hij enige ervaring had met het gebruik van een groot besturingssysteem, gezien vanuit het standpunt van de gebruiker.

De eerste twee hoofdstukken omschrijven de functies van een besturingssysteem en beschrijven enkele algemene eigenschappen van besturingssystemen. In hoofdstuk 3 wordt het basisbegrip 'proces' gevestigd voor de bespreking van gebeurtenissen die gelijktijdig in het besturingssysteem plaatsvinden; tevens wordt er besproken hoe processen met elkaar communiceren. Vervolgens beschrijft de rest van het boek de constructie van een besturingssysteem van binnen naar buiten (bottom up). We beginnen met de koppeling aan de machinehardware en we eindigen met de koppeling naar de gebruiker. Aan het einde van het boek zien we dat het geconstrueerde systeem alle eigenschappen bezit die we aan het begin hadden gevraagd.

Door het hele boek heb ik getracht te laten zien hoe iedere stap in de constructie van een besturingssysteem een logisch vervolg is op de vorige. Tevens heb ik de logische structuur van het gehele systeem benadrukt. Dit heb ik om twee redenen gedaan. De eerste is van pedagogische aard: mijn ervaring leert dat studenten een beter begrip ontwikkelen van complex materiaal indien dat op een

samenhangende wijze gepresenteerd wordt. De tweede is eerlijk gezegd polemisch: ik geloof dat dit de manier is waarop besturingssystemen geconstrueerd moeten worden. Aandacht voor de structuur en logische afhankelijkheid zijn de beste middelen die we hebben voor de bouw van besturingssystemen die gemakkelijk te begrijpen, gemakkelijk te onderhouden en relatief foutloos zijn.

Ten slotte zou ik graag de vele vrienden en collega's willen bedanken die bij het schrijven van dit boek tot hulp geweest zijn. In het bijzonder zou ik David Lyons, die een vruchtbare bron van ideeën en commentaar is geweest, willen bedanken; samen met David Howarth, die veel waardevolle opmerkingen heeft gemaakt op een vorig concept. Mijn dank gaat ook uit naar Colin Strutt, Morris Sloman, John Forecast, Ron Bushell en Bill Hart, die allen constructieve suggesties hebben gedaan voor het verbeteren van de tekst.

ANDREW LISTER

Voorwoord bij de tweede editie

Elk boek over informatica lijdt onder het feit dat het materie behandelt die aan snelle ontwikkeling onderhevig is, zowel wat de techniek als de principes betreft. Een boek over besturingssystemen is hier geen uitzondering op: de hardware waarop het loopt ontwikkelt zich stormachtig door de grote mate van integratie van onderdelen, en de ideeën over wat een besturingssysteem moet doen worden herzien door veranderende eisen en verwachtingen van de gebruiker. Alhoewel dit boek over de 'grondbeginselen' gaat, waarvan verwacht kan worden dat ze minder snel veranderen dan andere onderwerpen, zou het dwaas zijn voorbij te gaan aan de ontwikkelingen die hebben plaatsgevonden sinds de publicatie vier jaar geleden.

Daarom heb ik de tekst voor deze editie op drie manieren aangepast. Ten eerste is er een hoofdstuk over betrouwbaarheid bijgekomen, een onderwerp waarvan het belang dramatisch toegenomen is, omdat in steeds meer gebieden van het dagelijkse leven vertrouwd wordt op computersystemen. Ten tweede zijn de verwijzingen naar bestaande besturingssystemen bijgewerkt, zodat er nu ook verwezen wordt naar systemen die na de eerste editie in gebruik zijn genomen. Ten derde heb ik de presentatie van bepaalde onderwerpen zo veranderd, dat ze de tegenwoordig gangbare denkbeelden beter weergeven, en heb ik het laatste hoofdstuk uitgebreid om aan te geven welke ontwikkelingen in de toekomst verwacht kunnen worden.

Andere veranderingen in de tekst komen voort uit de reacties van lezers van de eerste editie. Op dit punt ben ik veel dank aan mijn studenten verschuldigd; zij merkten fouten op en maakten onduidelijkheden aan mij duidelijk. Alle verbeteringen op dat gebied zijn voornamelijk aan hen te danken, eventueel overgebleven onvolkomenheden zijn natuurlijk van mij.

A.L.

Voorwoord bij de derde editie

Bijna een tiental jaren na de eerste publicatie is het geruststellend te weten dat de grondbeginselen in essentie hetzelfde blijven. Omdat echter de herinnering aan de systemen uit het begin van de jaren '70 geleidelijk begon te verbleken, vond ik het noodzakelijk de tekst bij te werken door te verwijzen naar hun opvolgers. Ik heb ook van de gelegenheid gebruik gemaakt om een paar nieuwe onderwerpen te introduceren en de presentatie van diverse andere te verbeteren. Wat dit laatste betreft ben ik dank verschuldigd aan Morris Sloman en Dirk Vermeir en, zoals altijd, aan mijn studenten voor hun waardevolle suggesties.

A.L.

1 Inleiding

In de eerste twee hoofdstukken zullen we de volgende vragen trachten te beantwoorden:

Wat is een besturingssysteem?
Wat doet het?
Waarom hebben we het nodig?

Met het beantwoorden van deze vragen hopen wij de lezer een idee te geven van het onderwerp van de daaropvolgende hoofdstukken: de architectuur van besturingssystemen.

Eerst gaan we de functie van besturingssystemen in het algemeen bekijken en gaan wij de bestaande besturingssystemen in groepen indelen.

Algemeen gesproken kan men stellen dat een besturingssysteem de volgende twee hoofdfuncties verzorgt.

(1) Eerlijk en efficiënt verdelen van de faciliteiten

Bij gelijktijdig gebruik door meerdere personen moet het besturingssysteem de capaciteit van de computer verdelen tussen de gebruikers. Het doel is enerzijds het verhogen van de toegankelijkheid van de computer voor de gebruikers en anderzijds het gelijktijdig verzekeren van een optimaal gebruik van componenten als processoren, het geheugen en de in- en uitvoerapparatuur. Het belang van optimaal gebruik van een component hangt af van de kosten van de betreffende component; de steeds verminderende kosten van hardware hebben ertoe geleid dat er ook steeds minder nadruk komt te liggen op optimaal gebruik, zelfs in zo sterke mate dat vele microcomputers slechts een enkelvoudige toepassing hebben en dus nooit door meerdere mensen tegelijk gebruikt zullen worden. Dit is echter in tegenstelling tot grote computers, waarvan de hoge prijs ervoor zorgt dat er aanzienlijke moeite gedaan wordt om tot gemeenschappelijk gebruik te komen.

(2) Presentatie van een virtuele machine

De tweede hoofdfunctie van een besturingssysteem is het transformeren van een moeilijk bestuurbaar stuk hardware in een gebruikersvriendelijke machine. Men kan dit beschouwen alsof de gebruiker een *virtuele machine* ziet die anders, maar beter handelbaar is dan de fysieke machine. Hier volgen enkele gebieden waarin de virtuele machine vaak verschilt met de erachter liggende werkelijke machine.

(a) Input/Output (I/O)
De I/O mogelijkheden van de hardware kunnen zeer complex zijn en er zijn erg ingewikkelde programma's voor nodig om ze te benutten. Een besturingssysteem verlost de gebruiker van de noodzaak deze complexiteit te begrijpen en zorgt voor een virtuele machine die veel gemakkelijker te gebruiken is, maar even krachtige I/O mogelijkheden heeft.

(b) Geheugen
Veel besturingssystemen laten een virtuele machine zien waarvan het geheugen verschilt van de werkelijke machine. Bijvoorbeeld, een besturingssysteem kan secundaire geheugens gebruiken (in de vorm van schijven) om het hoofdgeheugen groter te laten lijken; het is echter ook mogelijk dat het besturingssysteem het hoofdgeheugen verdeelt tussen de gebruikers zodat elke individuele gebruiker een virtuele machine ziet waarvan het geheugen kleiner is dan dat van de werkelijke machine.

(c) Opslagsysteem
De meeste virtuele machines omvatten ook een *opslagsysteem* voor de lange-termijn opslag van programma's en gegevens. Het opslagsysteem is gebaseerd op opslag op schijf of band door de werkelijke machine, maar het besturingssysteem stelt de gebruiker in staat de opgeslagen informatie door middel van een symbolische naam te benaderen in plaats van benadering van de fysieke lokatie op het opslagmedium.

(d) Bescherming en foutbehandeling
Daar de meeste grote computersystemen door meerdere gebruikers gedeeld worden, is het essentieel dat iedere gebruiker tegen de gevolgen van fouten of kwade wil van anderen beschermd wordt. Tussen computers bestaan er aanzienlijke verschillen in de mate waarin door de hardware bescherming wordt geboden. Het besturingssysteem moet op de geboden beschermingsmogelijkheden voortborduren zodat er een virtuele machine ontstaat waarin de verschillende gebruikers elkaar niet kunnen beïnvloeden.

(e) Programma interactie.
Een virtuele machine kan voor faciliteiten ten behoeve van gebruikersprogramma's zorgen. Bijvoorbeeld zó dat de resultaten, verkregen uit één programma, met een ander programma verder verwerkt kunnen worden.

(f) Programmabesturing
Een virtuele machine voorziet de gebruiker van een aantal middelen om de programma's en data in de computer te manipuleren. De gebruiker heeft de beschikking over een *besturingstaal* die hem in staat stelt de virtuele machine op te dragen wat hij wil, bijvoorbeeld het vertalen en uitvoeren van een programma, het samenvoegen van twee gegevensverzamelingen die zich in het opslagsysteem bevinden, etcetera. De besturingstaal staat op een veel hoger niveau en is veel gemakkelijker te gebruiken dan de machinecode-instructies die door de fysieke machine uitgevoerd kunnen worden.

De precieze aard van de virtuele machine hangt af van het doel waarvoor zij gebruikt gaat worden. De eigenschappen van een machine voor het reserveren van vliegtuigplaatsen verschillen bijvoorbeeld van die van een machine die wetenschappelijke experimenten moet besturen. Het kan dan ook niet anders dan dat het ontwerp van het besturingssysteem sterk beïnvloed wordt door het soort gebruik waar de machine voor bedoeld is. Helaas is het met machines voor algemeen gebruik vaak zo dat het soort gebruik niet gemakkelijk vastgesteld kan worden; een veel voorkomende kritiek op vele systemen is dan ook dat de poging om iedereen helemaal tevreden te stellen tot gevolg heeft dat niemand helemaal tevreden is. In het volgende deel zullen wij verschillende soorten systemen onderzoeken en een aantal van hun eigenschappen vaststellen.

1.1 SOORTEN BESTURINGSSYSTEMEN

(1) Systemen voor één gebruiker (Single-user Systemen)

Single-user systemen verzorgen een virtuele machine voor, zoals al uit de naam blijkt, één gebruiker. Zij zijn geschikt voor computers die slechts op één moment voor één functie gebruikt worden, of die zo goedkoop zijn dat deel gebruik niet lonend is. De meeste besturingssystemen voor microcomputers (zoals CP/M en MS-DOS die op veel personal computers draaien) zijn van het Single-user type. Single-user systemen zorgen meestal voor een simpele virtuele machine die het mogelijk maakt om met een scala aan softwarepakketten te werken en die de gebruiker tevens in staat stellen zijn eigen programma's te ontwikkelen en uit te laten voeren. De meeste nadruk ligt op het presenteren van een gemakkelijk te gebruiken besturingstaal, een simpel opslagsysteem en in- en uitvoerfaciliteiten voor terminal en schijf.

(2) Procesbesturing

Generaliserend kan men zeggen dat de term procesbesturing slaat op het besturen van een industrieel proces door een computer, voorbeelden zijn olieraffinage en het fabriceren van machineonderdelen. Ruimer genomen kunnen zaken als klimaatbesturing in een ruimtecapsule, en het bewaken van de toestand van een patiënt in een ziekenhuis er ook onder verstaan worden. In al deze applicaties wordt er van *feedback* gebruik gemaakt; dat wil zeggen dat de computer gegevens (input) ontvangt van het bestuurde proces, de daarbij behorende respons berekent die de stabiliteit handhaaft, en het mechanisme aanstuurt om die respons te geven. Als bijvoorbeeld de inkomende gegevens een gevaarlijke temperatuurstijging aangeven, dan is het heel wel mogelijk dat de respons het openen van een klep is zodat er meer koelvloeistof gaat stromen. Vanzelfsprekend is er een kritische tijd waarbinnen de respons gegeven moet worden om het proces in evenwicht, en zeker veilig (!), te houden. De hoofdfunctie van een besturingssysteem bij procesbesturing is het geven van een maximale betrouwbaarheid met een minimum aan interventie van de operator (degene die de computer bedient), en ervoor te zorgen dat er bij hardwarestoringen geen ernstige gevolgen zijn (fail safe).

(3) Systemen voor het raadplegen van opgeslagen gegevens

Kenmerkend voor deze systemen is dat een grote hoeveelheid gegevens, een gegevensbestand of *database* voor informatie geraadpleegd kan worden. De responstijd op een informatieverzoek moet kort zijn (normaliter minder dan een minuut). Tevens moet de database veranderd kunnen worden als de informatie bijgewerkt wordt. Voorbeelden zijn management-informatiesystemen, waarbij de informatie bestaat uit gegevens over de gang van zaken in een bedrijf, en medische informatiesystemen, waar de gegevens van de patiënten het bestand vormen. De gebruiker (bedrijfsleider of dokter) gaat ervan uit dat hij de informatie kan verkrijgen zonder dat hij weet hoe het bestand, hard- of softwarematig, werkt. Het besturingssysteem moet het dus mogelijk maken informatie op te vragen zonder dat de gebruiker betrokken wordt bij de details van de implementatie.

(4) Het verwerken van transacties

Systemen voor het verwerken van transacties worden gekarakteriseerd door een database die veelvuldig, soms wel enkele keren per seconde, veranderd wordt. Duidelijke voorbeelden hiervan zijn te vinden bij het reserveren van vliegtuigplaatsen en bij het bankieren. In het eerste geval bevat het bestand informatie over de beschikbaarheid van plaatsen, informatie die bij elke boeking wordt aangepast. In het tweede geval bestaat het bestand uit gegevens ove..

rekeningen die bij elke debet- of creditboeking aangepast worden. De grootste nadruk bij het verwerken van transacties ligt op het direct bijhouden van het bestand. Het zal duidelijk zijn dat het systeem onbruikbaar is als er gemuteerd wordt op foutieve gegevens. Ook doen er zich extra problemen voor als er tegelijkertijd meerdere mutaties op dezelfde gegevens plaatsvinden (denk hierbij aan twee reisagenten die proberen dezelfde stoel te boeken). Iedere individuele gebruiker moet zich natuurlijk niet met deze problemen bezig gaan houden en het besturingssysteem moet dit zo oplossen dat iedere eindgebruiker het gevoel heeft dat hij de enige gebruiker is.

(5) Universele systemen

Universele systemen worden toegepast bij computers die vele gebruikers bedienen die een heel scala aan toepassingen uitvoeren. Zulke systemen worden ontworpen voor het verwerken van een aanhoudende stroom opdrachten (jobs) die op de computer uitgevoerd moeten worden. Iedere job voert een specifieke taak uit voor een bepaalde gebruiker (bijvoorbeeld de analyse van meetwaarden, het oplossen van differentiaalvergelijkingen en het berekenen van de maandelijkse salarissen) en bestaat meestal uit het uitvoeren (runnen) van een of meerdere programma's. In een eenvoudig geval, bijvoorbeeld dat van de meetwaarden, kan de opdracht slechts bestaan uit het uitvoeren van een al vertaald programma met een bepaalde set gegevens. Ingewikkelder opdrachten kunnen het gebruik van een 'editor' voor het veranderen van een programma vereisen, waarna dit vertaald en daarna pas uitgevoerd kan worden. Gezien de variëteit in de opdrachten die gegeven kunnen worden moet het systeem over vele functieprogramma's (programma's die het werken met de computer vereenvoudigen) beschikken. Voorbeelden van deze functieprogramma's zijn vertaalprogramma's (compilers) voor diverse computertalen, assemblers, editors, debugging (foutzoek) pakketten, tekstverwerkers en een opslagsysteem voor de permanente opslag van informatie. De computer moet ook in staat zijn te werken met de grote variëteit aan randapparatuur die vereist kan zijn (bijvoorbeeld kaartlezers en ponsers, terminals, regeldrukkers, band- en schijfopslag apparatuur, grafische plotters). Het voorzien in en het besturen van deze faciliteiten, samen met de organisatie van de stroom werk, zijn, in hoofdlijnen, de functies van een besturingssysteem voor algemeen gebruik.

Systemen voor algemeen gebruik kunnen in twee groepen ingedeeld worden: (1) *batch*, en (2) *multi-access*. Het hoofdkenmerk van een batch-systeem (stapelsgewijs) is dat vanaf het moment dat een opdracht de computer ingaat tot aan het moment dat de computer klaar is, de gebruiker geen invloed op de opdracht heeft. Een bepaalde opdracht wordt in kaarten geponst of op magnetische band opgeslagen en wordt dan aan een operator gegeven die deze in de computer invoert. Als de opdracht uitgevoerd is, stuurt de operator de uitvoer terug naar de gebruiker. Meestal wordt de opdracht aan de computer doorgegeven met behulp van in- en uitvoerapparatuur

die in de computerruimte zelf staat opgesteld. Bij sommige batch-systemen met RJE (*remote job entry; invoer op afstand*) is het echter mogelijk om opdrachten aan de computer te geven met I/O (Input/Output) apparatuur die op afstand van de centrale installatie staat opgesteld en verbonden is door een datatransmissielijn. Een RJE-systeem kan meerdere op afstand gelegen I/O stations ondersteunen en moet de uitvoer natuurlijk naar het station sturen dat de daarbij behorende opdracht gaf. Bij een batch- of een RJE-systeem kan de gebruiker niet ingrijpen in zijn opdracht op het moment dat die uitgevoerd wordt.

In tegenstelling hiermee kan bij een multi-access systeem (meervoudige toegang) de gebruiker een opdracht op een terminal invoeren, bekijken en besturen tijdens de uitvoering hiervan. Hij kan bijvoorbeeld spellingsfouten, die door het vertaalprogramma gesignaleerd worden, corrigeren of gegevens invoeren die afhankelijk zijn van de tot dan verkregen resultaten. Het besturingssysteem verdeelt de beschikbare computerfaciliteiten zo tussen de diverse opdrachten, alsof het lijkt of iedere individuele gebruiker de machine helemaal voor zichzelf alleen heeft.

Veel besturingssystemen combineren de beide werkmethodes. Vanwege de voordelen van interactie wordt de multi-access methode meestal gebruikt voor zaken als programma-ontwikkeling en het maken van documenten, terwijl de batch-verwerking alleen maar gebruikt wordt voor niet interactieve routine opdrachten zoals het maken van salaris- en voorraadlijsten.

Veel van de bovengenoemde systemen kunnen zowel op een enkele computer als op een groep, onderling verbonden, computers geïmplementeerd worden. In het tweede geval kunnen de afzonderlijke computers het totale opdrachtenpakket onderling gelijk verdelen, of kan iedere computer afzonderlijk een bepaald aspect eruit toegewezen krijgen. Van alle stations kunnen bijvoorbeeld alle I/O handelingen aan een computer toegewezen worden terwijl een andere computer al het intensieve rekenwerk doet. De computers kunnen óf in dezelfde ruimte staan en mogelijk zelfs geheugen- en opslagcapaciteit delen, óf op enige afstand van elkaar staan en communiceren via datalijnen. In zulke *gedecentraliseerde systemen* moet het besturingssysteem de activiteiten van de afzonderlijke computers coördineren en ervoor zorgen dat de juiste informatie op de juiste momenten tussen de computers verzonden wordt.

1.2 HET 'PAPIEREN' BESTURINGSSYSTEEM

Het grootste deel van dit boek zal gewijd zijn aan systemen voor algemeen gebruik. Sommige van de besproken problemen doen zich echter ook voor bij andere systemen en waar dat van toepassing is, zullen wij het belang van onze bespreking ook daarvoor aangeven.

Om een besturingssysteem voor algemeen gebruik te kunnen bestuderen zullen wij een 'papieren' besturingssysteem gaan maken. Dat wil zeggen dat we een hypothetisch systeem op papier zullen ontwikkelen zodat we hopelijk de erbij betrokken principes kunnen laten zien. Het is onze bedoeling dat we dit systeem op een logische manier gaan ontwikkelen, dat wil zeggen dat iedere stap een natuurlijk vervolg zal zijn op de vorige. De uiteindelijke structuur zal in bepaalde opzichten op een ui lijken, waarbij iedere laag een stel functies verzorgt die slechts afhankelijk zijn van de daarbinnen liggende lagen. Het ontwerpen zal van binnen naar buiten gaan (bottom up), zodat we bij de kern zullen beginnen en daar lagen aan gaan toevoegen voor geheugenbeheer, I/O besturing, toegang tot opslag, enzovoorts.

Toegegeven moet worden dat er in werkelijkheid slechts een paar besturingssystemen een zo nette gelaagdheid ten toon spreiden; voorbeelden, die het opmerken waard zijn, zijn T.H.E. (Dijkstra, 1968), RC-4000 (Hansen, 1970), UNIX (Ritchie en Thompson, 1974) en in de commerciële wereld: VME voor de ICL 2900 serie (Keedy, 1976, Huxtable en Pinkerton, 1977). Men kan zelfs zonder problemen stellen dat veel bestaande systemen überhaupt weinig logische structuur tonen. Dit treurige feit kan aan drie factoren toegeschreven worden. Ten eerste werden de meeste, vandaag de dag gebruikte systemen ontworpen op het moment dat de principes voor het construeren van besturingssystemen veel minder duidelijk waren dan tegenwoordig. Ten tweede resulteert het laten werken van veel mensen aan complexe software, wat gebruikelijk was, zelden in een gaaf produkt. En ten derde leidde de al eerder genoemde wens, het maken van een systeem dat alles voor iedereen in zich heeft, vaak tot het *ad hoc* aankoppelen van extra mogelijkheden, soms tot ver na de eerste ontwerpfase. Verder is het ook nog zo dat er, onder dwang van eigenaardigheden in het hardware-ontwerp, een goed gestructureerd ontwerp verfomfaaid wordt bij de implementatie. We kunnen echter redelijkerwijs de hoop koesteren dat deze ongemakken steeds minder worden naarmate de hardware meer ontworpen wordt om aan te sluiten bij de constructie van besturingssystemen.

Gedurende de ontwikkeling van ons papieren besturingssysteem zullen we het vergelijken met bestaande systemen en zullen we de door ons gebruikte technieken vergelijken met die, welke elders gebruikt worden. Op die manier houden we ons besturingssysteem gegrondvest op de alledaagse praktijk, terwijl we het tegelijkertijd gebruiken als een voertuig voor het overbrengen van de principes van de constructie van besturingssystemen.

2 Functies en karakteristieken van besturingssystemen

In dit hoofdstuk zullen we wat verder ingaan op de functies waarvan we verwachten dat ze door het besturingssysteem verzorgd worden. Ook zullen we er enkele karakteristieken uitlichten die het moet bezitten om die functies te vervullen.

2.1 FUNCTIES VAN BESTURINGSSYSTEMEN

We gaan uit van de kale machine - de basis computerhardware, bestaande uit centrale processor, geheugen en diverse randapparatuur. (Voorlopig laten we computers met meer dan een centrale processor buiten beschouwing.) Bij afwezigheid van softwarehulp moet het hele proces van het laden en laten uitvoeren van een gebruikersprogramma met de hand worden gedaan. Als we van batchverwerking uitgaan, moet de operator een monnikkenwerkje verrichten, ongeveer zoals dat hieronder staat:

(1) Plaats de kaarten met het bronprogramma in de kaartlezer (of een ander invoerapparaat)
(2) Start een programma om de kaarten in te lezen
(3) Start een vertaalprogramma om het bronprogramma te vertalen
(4) Plaats de kaarten met de gegevens, als die er zijn, in de kaartlezer
(5) Start de uitvoering van het gecompileerde (vertaalde) programma
(6) Haal de resultaten van de regeldrukker.

De snelheid van de machine is duidelijk afhankelijk van het tempo waarin de operator de knoppen kan bedienen en de randapparatuur kan voorzien van gegevens; het systeem zou je dus operator-beperkt kunnen noemen.

Een voor de hand liggende eerste verbetering is de automatisering van de serie stappen die nodig zijn voor het inlezen, vertalen, laden en laten uitvoeren van programma's. De rol van de operator beperkt zich dus tot het laden van de kaarten aan de ene kant van de machine, en het afscheuren van het papier aan de andere kant,

de doorvoersnelheid wordt als gevolg hiervan vergroot. Voor niets gaat alléén de zon op en de prijs die we voor de verhoogde doorvoersnelheid moeten betalen is dan ook, dat een deel van de machine toegewezen wordt aan een programma dat de opeenvolgende handelingen bestuurt die voor het karwei nodig zijn. Daar niet bij alle klussen dezelfde volgorde van handelen nodig is (soms is het bijvoorbeeld niet nodig te vertalen) moet het programma in staat zijn uit een of meer besturingskaarten op te maken welke handelingen er in welk geval nodig zijn. Anders gezegd, een besturingsprogramma moet een *job control language (karwei besturingstaal)* kunnen begrijpen. Verder mogen fouten die onvermijdelijk bij sommige opdrachten ontstaan geen effect hebben op opdrachten die later in de wachtrij staan. Een besturingsprogramma is dus verantwoordelijk voor het rationeel oplossen van ontstane fouten. Een besturingsprogramma met deze eigenschappen is feitelijk een besturingssysteem in zijn meest beknopte vorm.

Een dergelijk systeem is duidelijk beperkt door de snelheid waarmee het in- en uitvoer kan verzorgen en de volgende stap om de efficiëntie te verbeteren is het verminderen van de afhankelijkheid van die I/O. Historisch gezien werd dit het eerst bereikt met *off-lining (loskoppeling)*, hierbij werd alle I/O op magnetische band gedaan. Het omzetten van kaart- (of papierband-) invoer naar magnetische band, en van magneetbanduitvoer naar de regeldrukker of een ponser, werd verbannen naar een minder krachtige en goedkopere, losstaande computer. Het overzetten van de magneetbanden tussen de hoofd- en bijcomputer ging met de hand. Losgekoppelde systemen, in het bijzonder de IBM Fortan Monitor System, voldeden goed van het einde van de jaren vijftig tot halverwege de jaren zestig.

Het loskoppelen verminderde de afhankelijkheid van de I/O door ervoor te zorgen dat alle I/O op magneetband plaatsvond, destijds het snelste, goedkoop beschikbare, medium. De prijs die hiervoor betaald werd was dat er aan het heel simpele besturingssysteem een set programmainstructies (routines) toegevoegd moest worden, voor het coderen en opslaan op band van de gegevens, die uiteindelijk voor een ander uitvoerapparaat bestemd waren (en omgekeerd voor de invoer). Er moet ook nog op gewezen worden dat, omdat band in principe een serieel medium is, er geen neiging bestond om opdrachten in een andere dan op de band gezette volgorde te laten uitvoeren.

Om de afhankelijkheid van I/O te elimineren in plaats van alleen maar te reduceren, moeten er technieken toegepast worden waarbij de I/O en de verwerking elkaar overlappen. Met behulp van twee hardwarehulpmiddelen is dat mogelijk, te weten: de *kanaal (transmissiekanaal)* en de *interrupt (ingreepsignaal)*. Een kanaal is een hulpmiddel dat een of meer randapparaten, onafhankelijk van de centrale processor, bestuurt door het heen en weer sturen van gegevens tussen de randapparatuur en het geheugen. Een interrupt is een signaal dat de besturing van de centrale processor naar een vaste (geheugen-) plaats stuurt, terwijl tegelijkertijd de laatste waarde van de programmateller opgeslagen wordt. Dit houdt dus in

dat het programma dat uitgevoerd wordt, tijdelijk verlaten wordt, maar later weer vervolgd kan worden (*geïnterrumpeerd* wordt). Een interrupt van een kanaal werkt dus als een signaal dat aangeeft dat de gegevensoverdracht klaar is. Zodoende is het mogelijk dat de centrale processor de gegevensoverdracht naar randapparatuur start, dan doorgaat met de verwerking terwijl het kanaal de overdracht bestuurt, en als de overdracht klaar is daarvan op de hoogte gesteld wordt door een interrupt. (De geïnteresseerde lezer die meer wil weten over de hardware van interrupts en kanalen verwijzen wij naar boeken over computerarchitectuur zoals die van Foster, 1970, of van Stone, 1975.)

We hebben nu de mogelijkheid om opdrachten op een geschikt medium op te slaan, meestal schijf, en ze stuk voor stuk uit te voeren op het moment dat andere ingelezen worden. De uitbreidingen aan ons snelgroeiende besturingssysteem zijn: ten eerste een routine om de interrupts te verwerken en ten tweede een routine die bepaalt welke van de op schijf staande opdrachten als volgende uitgevoerd moet worden. De tweede functie, het *maken van de werkindeling (scheduling)*, is een gevolg van het gebruik van een schijf als invoermedium in plaats van een band; de schijf is willekeurig (random) toegankelijk terwijl de band in volgorde (serieel) afgewerkt moet worden. Een aldus werkend systeem heet een *single stream batch monitor*. Single stream wil zeggen dat er slechts één opdracht tegelijk wordt uitgevoerd. Dit waren de meest voorkomende systemen halverwege de jaren '60.

Het grootste nadeel van een single stream systeem is dat het maar één karwei tegelijkertijd aankan, onafhankelijk van de grootte ervan. Dit nadeel is te ondervangen door *multiprogrammering* - het simultaan laten uitvoeren van meerdere programma's op een machine. Het idee hierachter is dat er tegelijkertijd meerdere programma's in het geheugen gehouden worden en dat de centrale processor zijn tijd hiertussen verdeelt, afhankelijk van de hulpbronnen (zoals transmissiekanalen of randapparatuur) die elk van de programma's op een gegeven ogenblik nodig heeft. Zodoende is het mogelijk om constant een geschikte mengeling van opdrachten in de machine te hebben, met als resultaat dat de faciliteiten optimaal gebruikt worden. Dit geldt ook voor het gebruik van de centrale processor, steeds als een opdracht wacht op in- of uitvoer kan de processor overschakelen naar een andere opdracht die op dat moment in de machine aanwezig is. De extra last voor het besturingssysteem is de besturing van de hulpbronnen en de bescherming van het ene karwei tegen de activiteiten van het andere. Een dergelijk besturingssysteem heet een *multi-stream batch monitor*, versies hiervan zag men veel toegepast op grote computers in het begin van de jaren '70 (bijvoorbeeld GEORGE3 op de ICL 1900 serie, en OS/360 op de 360 en 370 series van IBM. De ontwikkeling van deze systemen had veel te danken aan het pionierswerk dat in het begin van de jaren '60 op de Atlas computer van de Manchester Universiteit gedaan werd.

Op dit punt in de ontwikkeling van onze kale machine hebben we een behoorlijk fijn ontwikkeld systeem dat goed van de beschikbare hardware gebruik maakt. Het grootste nadeel, gezien vanuit het standpunt van de gebruiker, is het ontbreken van de interactie met zijn programma als dat door de computer uitgevoerd wordt. Interactie is in veel situaties waardevol, in het bijzonder bij het ontwikkelen en debuggen (corrigeren van fouten) van een programma. Het debuggen wordt bijvoorbeeld een stuk eenvoudiger als de programmeur zijn veranderingen direct kan testen, er de gevolgen van kan bekijken en zo nodig verdere wijzigingen kan aanbrengen. Op dezelfde manier hebben veel programma's voor het oplossen van probleemstellingen 'sturing' van de gebruiker nodig, dat wil zeggen dat de gebruiker het verdere verloop van het programma bepaalt aan de hand van de tot dan verkregen resultaten. Wil interactie bruikbaar zijn dan moet het multi-stream batch systeem zo aangepast worden dat het invoer accepteert van gebruikers van invoerstations die op afstand gelegen zijn. Anders gezegd, het moet een multi-access systeem worden zoals in het vorige hoofdstuk besproken is. In het begin van de jaren '70 maakten de meeste fabrikanten multi-access besturingssystemen voor hun grote computers, zoals MINIMOP (ICL 1900), TSS/360 (IBM 360 en 370) en TOPS-10 (DEC System-10). In het midden van de jaren '70 werden de batch en multi-access methode gecombineerd in systemen als MVS (IBM 370), MCP (Burroughs 6700 en 7700) en VME (ICL 2900).

Het bovenstaande heeft ons gevoerd naar een systeem dat ten minste de volgende functies moet kunnen vervullen of hebben:

(1) bepalen van de opdrachtvolgorde
(2) het interpreteren van de besturingstaal voor de opdrachten
(3) het behandelen van fouten
(4) het verzorgen van in- en uitvoer
(5) het behandelen van interrupts
(6) het maken van werkindelingen
(7) hulpbronbesturing
(8) bescherming
(9) multi-access

Verder moet het besturingssysteem gemakkelijk werken gezien vanuit het gezichtspunt van de gebruiker en moet het gemakkelijk te besturen zijn gezien vanuit het gezichtspunt van degene die verantwoordelijk is voor de installatie. We kunnen dus de volgende functies toevoegen:

(10) het voorzien in een goede interface voor degene die de computer bedient
(11) het voor zijn rekening nemen van computerhulpbronnen/faciliteiten.

Hieruit blijkt wel dat besturingssystemen beslist niet onbeduidend zijn en het is dan ook niet verwonderlijk dat er heel wat moeite

gedaan is om ze te schrijven. Als een stap naar een coherent besturingssysteem kunnen we uit de bovenstaande lijst van functies karakteristieken afleiden die het systeem ten toon moet spreiden. Die worden in het volgende deel opgesomd, samen met, ter illustratie, de erbij behorende problemen.

2.2 DE KARAKTERISTIEKEN VAN EEN BESTURINGSSYSTEEM

(1) Gelijktijdigheid (concurrency)

Gelijktijdigheid is het aanwezig zijn van meerdere opeenvolgende of parallelle activiteiten. Voorbeelden hiervan zijn het overlappen van in- en uitvoer met gegevensverwerking, en het naast elkaar in het geheugen aanwezig zijn van meerdere gebruikersprogramma's. Gelijktijdigheid brengt het probleem met zich mee dat er van de ene naar de andere activiteit geschakeld moet worden, waarbij de ene activiteit tegen de effecten van de andere beschermd moet worden, en waarbij activiteiten gelijkgeschakeld moeten worden daar waar zij van elkaar afhankelijk zijn.

(2) Medegebruik

Van tegelijkertijd lopende activiteiten mag verwacht worden dat zij dezelfde computerhulpbronnen en informatie gebruiken. De reden hiervoor is viervoudig.

(a) Kosten: het is overdreven om voor alle gebruikers apart voldoende faciliteiten te scheppen.
(b) Voortborduren op het werk van anderen: het is nuttig om de programma's en routines van anderen te kunnen gebruiken.
(c) Gemeenschappelijk gebruik van gegevens: soms kan het nodig zijn om hetzelfde gegevensbestand te gebruiken voor diverse andere programma's, die mogelijk door verschillende gebruikers toegepast worden.
(d) Het verwijderen van overtolligheden: het is economisch verantwoord om een kopie van een programma (bijvoorbeeld een vertaalprogramma) te delen met meerdere gebruikers; dit in plaats van het voor ieder apart aanschaffen van een kopie.

Problemen die samenhangen met het delen van faciliteiten en hulpbronnen zijn: toewijzing van hulpmiddelen, gelijktijdige toegang tot gegevens, gelijktijdige programma-uitvoering en bescherming tegen fraude.

(3) 'Permanente' of niet vluchtige opslag

De behoefte programma's te delen betekent dat er een permanente opslag van informatie nodig is. Permanente opslag geeft de gebruiker ook het gemak dat hij zijn programma's of gegevens in de computer kan opslaan in plaats van op een extern medium (bijvoorbeeld kaarten). De problemen die hieruit voortvloeien zijn: voorzien in gemakkelijke toegankelijkheid, bescherming tegen gebruik door anderen (al dan niet kwaadwillig) en bescherming tegen storingen in het systeem.

(4) Flexibele respons (nondeterminacy)

De besluitvorming van een besturingssysteem moet in een bepaald opzicht altijd *van tevoren bepaald zijn*; als vandaag of morgen hetzelfde programma met dezelfde gegevens uitgevoerd wordt dan moeten er altijd dezelfde resultaten uit komen. In een ander opzicht moet de besluitvorming *niet van tevoren bepaald zijn* als het moet reageren op gebeurtenissen die in een onvoorspelbare volgorde plaatsvinden. Te denken valt aan zaken als communicatie met hulpbronnen, fouten tijdens de uitvoering van programma's en onderbrekingen van randapparatuur. Gezien het enorme aantal willekeurige gebeurtenissen dat er kan plaatsvinden, is het duidelijk dat er niet verwacht kan worden dat er een besturingssysteem geschreven wordt, dat elk van die mogelijkheden dekt. In plaats daarvan moet het systeem zo geschreven worden dat het *elke* serie gebeurtenissen aankan.

Opgemerkt moet worden dat geen van deze karakteristieken specifiek slaat op systemen voor algemeen gebruik. Zo is bijvoorbeeld de permanente opslag vanzelfsprekend nodig bij gegevensinformatiesystemen, en gelijktijdigheid is een hoofdeigenschap van systemen voor het verwerken van transacties.

Tot slot van dit hoofdstuk noemen we kort wat eigenschappen die we wenselijk vinden bij een besturingssysteem voor algemeen gebruik.

2.3 WENSELIJKE EIGENSCHAPPEN

(1) Efficiëntie

Op de behoefte aan efficiëntie is al gewezen. Helaas is het moeilijk een enkel criterium aan te wijzen waarop de efficiëntie van een besturingssysteem beoordeeld kan worden: een aantal criteria zijn hieronder opgesteld:

(a) de gemiddelde tijd tussen twee opdrachten
(b) de tijd dat de centrale processor ongebruikt is
(c) uitvoertijd voor batch opdrachten
(d) responstijd (bij multi-access systemen)
(e) het gebruik van hulpbronnen
(f) verwerkingssnelheid (aantal opdrachten per uur)

Er kan niet tegelijkertijd aan al deze criteria voldaan worden; we zullen hier meer over zeggen in volgende hoofdstukken die over ontwerpbeslissingen handelen, ontwerpbeslissingen die aan bepaalde criteria meer gewicht geven dan aan andere.

(2) Betrouwbaarheid

In het ideale geval is een besturingssysteem helemaal vrij van fouten en kan het alle onverwachte gebeurtenissen aan. In de praktijk is dit echter nooit zo, maar in hoofdstuk 10 zullen we zien hoe we een aardig eind kunnen komen.

(3) Onderhoudbaarheid

Het moet mogelijk zijn een besturingssysteem te onderhouden - het aan te passen en fouten erin te verbeteren - zonder dat er een legertje programmeurs voor nodig is. Het logische gevolg hiervan is dat het systeem modulair van opbouw moet zijn met duidelijk gedefinieerde koppelingen tussen de modulen, en dat het goed gedocumenteerd moet zijn.

(4) Kleine omvang

De ruimte die nodig is om het besturingssysteem in op te slaan is verloren voor de produktieve verwerking. Verder zal een groot systeem waarschijnlijk vatbaarder zijn voor fouten en zal het meer tijd voor het schrijven vergen dan een klein systeem.

Dit hoofdstuk samenvattend: we hebben de algemene functies besproken waarvan we verwachten dat ze door een besturingssysteem worden vervuld; we hebben er enkele cruciale karakteristieken van besturingssystemen uitgelicht; en we hebben enkele wenselijke eigenschappen en mogelijkheden opgesomd. In het volgende hoofdstuk zullen we enkele eenvoudige instrumenten onderzoeken die ons bij het construeren van het gewenste besturingssysteem van dienst kunnen zijn.

3 Gelijktijdig lopende processen

Voordat we besturingssystemen nader kunnen bestuderen, moeten we wat basisbegrippen introduceren en enkele 'gereedschappen' ontwikkelen. Daar zullen we dit hoofdstuk voor gebruiken.

3.1 PROGRAMMA'S, PROCESSEN EN PROCESSOREN

Om te beginnen beschouwen we een besturingssysteem als een set activiteiten, die elk één van de functies behandelen die in hoofdstuk 2 beschreven zijn, zoals het maken van een werkindeling en I/O behandeling. Elke activiteit bestaat uit het uitvoeren van een of meer programma's en wordt steeds aangeroepen als de bijbehorende functie nodig is. We gebruiken het woord *proces* om een dergelijke activiteit te beschrijven. (Andere in de literatuur voorkomende namen zijn *task (taak)* en *computation (verwerking)*. Zodoende kan men zich een proces voorstellen als een serie acties, die als gevolg van een serie instructies (een programma) uitgevoerd wordt en waarvan het netto resultaat is, dat er voorzien wordt in een systeemfunctie. Het begrip kan zo uitgebreid worden, dat het zowel de gebruikers- als de systeemfuncties omvat, zodat het uitvoeren van een gebruikersprogramma ook een proces heet.

Een proces kan het uitvoeren van meer dan één programma inhouden; maar omgekeerd kan het ook zo zijn dat een programma of routine betrokken is bij meer dan één proces. Bijvoorbeeld, een routine voor het toevoegen van een onderdeel aan een lijst zou bij elk proces gebruikt kunnen worden waar er met wachtrijen gewerkt wordt. De wetenschap dat op een gegeven ogenblik een bepaald programma uitgevoerd wordt zegt ons dientengevolge niet veel over welke activiteit er uitgevoerd wordt of over welke functie er geïmplementeerd wordt. Voornamelijk hierom is het nuttiger begrip te hebben van een proces, dan begrip te hebben van een programma, als men besturingssystemen wil bespreken.

Een proces kan verlopen dankzij een werktuig dat het bijbehorende programma uitvoert. Dit werktuig is een *processor*. We zeggen dat een processor een proces *uitvoert* (Engels: *runs*) of dat een proces door een processor *uitgevoerd wordt*. Een processor is een

ding dat instructies uitvoert; afhankelijk van de aard van de instructies kan de processor in alleen hardware of in een combinatie van hard- en software geïmplementeerd worden. Zo is bijvoorbeeld een Centrale Verwerkings Eenheid (CVE) een processor voor het uitvoeren van machinetaalinstructies, terwijl de combinatie CVE/BASIC-interpretator een processor kan vormen voor het uitvoeren van BASIC instructies. Het omzetten van de kale computer in een virtuele machine, zoals in het eerste hoofdstuk is besproken, houdt in feite het combineren in van computerhardware met een besturingssysteem, om zo een processor te scheppen die in staat is gebruikersprocessen uit te voeren (dat wil zeggen, een processor die in staat is gebruikersinstructies uit te voeren).

Het programma of de programma's die te maken hebben met een proces hoeven niet altijd als software geïmplementeerd te worden. Zo kan bijvoorbeeld het werk van een transmissiekanaal - het verzorgen van gegevensoverdracht - beschouwd worden als een proces waarvan het bijbehorende programma is ingebakken in de hardware van het transmissiekanaal. Zo bekeken is het transmissiekanaal, of welke randapparatuur dan ook, een processor die slechts één proces kan uitvoeren.

De begrippen proces en processor kunnen nu zonder meer gebruikt worden om de in het vorige hoofdstuk besproken karakteristieken van het besturingssysteem, gelijktijdigheid en flexibele respons, te verklaren.

Gelijktijdigheid kan beschouwd worden als het tegelijkertijd activeren van diverse processen (dat wil zeggen, het uitvoeren van diverse programma's). Op voorwaarde dat er evenveel processoren als processen zijn levert dit geen logische problemen op. Echter als er, zoals gebruikelijk, minder processoren dan processen zijn, dan kan er 'schijngelijktijdigheid' verkregen worden door de processoren van het ene naar het andere proces te laten omschakelen. Als het omschakelen snel genoeg plaatsvindt, zal het systeem in een grotere tijdsschaal gezien het idee geven dat het gelijktijdig verwerkt. Schijngelijktijdigheid wordt zelfs door het laten afwisselen van processen op één processor bereikt.

Het trekken van een parallel met het werk van een secretaresse op een gewoon kantoor kan hier verhelderend zijn. Elke werkzaamheid die de secretaresse moet uitvoeren, het typen van brieven, het opbergen van rekeningen, of het notuleren van tekst, kan vergeleken worden met een proces in een besturingssysteem. De processor is de secretaresse zelf en de volgorde van de instructies die elke werkzaamheid definiëren is analoog met een programma. Als het druk is op het kantoor dan kan de secretaresse de ene werkzaamheid afbreken om een andere te gaan doen en in die situatie zou ze waarschijnlijk klagen dat ze 'diverse dingen tegelijk doet'. Natuurlijk is ze maar met één klus tegelijk bezig, maar het regelmatig overschakelen tussen de diverse klussen geeft een algemene indruk van gelijktijdigheid. De analogie kan verder door getrokken worden door op te merken dat de secretaresse goed moet vastleggen waar ze op dat ogenblik mee bezig is voordat ze naar iets anders kan overschakelen,

dit om ervoor te zorgen dat ze later de draad weer kan oppakken. Overeenkomstig hiermee moet een processor in een computersysteem gegevens vastleggen over het proces dat hij uitvoert voordat hij naar een ander proces kan overschakelen. De precieze aard van de informatie die opgeslagen wordt, zal in het volgende hoofdstuk besproken worden; voor het moment merken we op dat het voldoende moet zijn om het proces later weer te kunnen hervatten.

Als we de analogie nog verder doorvoeren, dan kunnen we een aantal extra secretaresses (processors) op kantoor aannemen. Echte gelijktijdigheid kan nu tussen de diverse secretaresses plaatsvinden, terwijl er zich ook nog schijngelijktijdigheid kan blijven voordoen bij de individuele secretaresses, als die tussen de ene en de andere klus heen en weer schakelen. Slechts als het aantal secretaresses gelijk is aan het aantal klussen dat uitgevoerd moet worden kan dat echt gelijktijdig gebeuren. De analogie met de processor die maar één proces kan uitvoeren, zoals een I/O apparaat, ligt in de vergelijking met de jongste bediende die alleen maar thee kan zetten.

Met het voorgaande in gedachten, omschrijven we *gelijktijdige (parallelle) verwerking* zo, dat het betekent dat, als we een momentopname maken van het gehele systeem, er zich meerdere processen tussen start en eindpunt van de uitvoering kunnen bevinden. Deze definitie omvat duidelijk zowel echte gelijktijdigheid als schijngelijktijdigheid.

Een flexibele respons is de tweede karakteristiek van een besturingssysteem dat zich gemakkelijk in termen van processen laat omschrijven. Als we een proces beschouwen als een serie acties, die tussen elke stap onderbroken kan worden, dan is een flexibele respons weerspiegeld in de niet te voorspellen volgorde waarin de onderbrekingen kunnen plaatsvinden, en dus ook in de onvoorspelbare volgorde waarin de handelingen verder gaan. Om terug te gaan naar de analogie met onze secretaresse, kunnen we de onvoorspelbare gebeurtenissen in een besturingssysteem vergelijken met het overgaan van telefoons op kantoor. Van tevoren kan men niet weten welke telefoon er precies over zal gaan, hoe lang men met het gesprek bezig zal zijn, of wat het effect van het gesprek zal zijn op de diverse activiteiten in het kantoor. We kunnen echter opmerken dat de secretaresse, die de telefoon opneemt, moet noteren wat ze op dat opgenblik aan het doen is, zodat ze er later mee door kan gaan. Hiermee overeenkomstig moet het onderbreken van een proces vergezeld gaan van het vastleggen van informatie die het later mogelijk moet maken dat proces weer te hervatten. De vastgelegde informatie is dezelfde als die welke nodig is bij het heen en weer schakelen van processoren tussen processen bij de gelijktijdigheid. Het is zelfs zo, dat we het onderbreken van een proces kunnen beschouwen als een omschakeling, die door een niet te voorziene gebeurtenis veroorzaakt wordt. En wel zo, dat de processor van het proces losgeschakeld wordt.

We kunnen hetgeen tot nu toe gezegd is als volgt samenvatten. Een proces is een serie acties en dus dynamisch, dit in tegenstelling tot een programma dat uit een serie instructies bestaat en dus

statisch is. Een processor is een middel om een proces uit te voeren. De flexibele respons en de gelijktijdigheid kunnen omschreven worden in termen van het onderbreken van processen tussen de acties door en het heen en weer schakelen van processoren tussen processen. Om onderbrekingen en omschakelingen uit te kunnen voeren moet er voldoende informatie opgeslagen zijn over een proces om later verder mee te kunnen gaan.

3.2 COMMUNICATIE TUSSEN PROCESSEN

De processen die in een computersysteem uitgevoerd worden staan natuurlijk niet op zichzelf. Aan de ene kant moeten ze samenwerken om het gestelde doel te bereiken: het uitvoeren van een gebruikersprogramma. En aan de andere kant zijn ze in competitie voor de beperkte faciliteiten en hulpmiddelen zoals processoren, geheugen en bestanden. Deze elementen, samenwerking en competitie, veroorzaken een behoefte aan een vorm van communicatie tussen de processen. De gebieden waar communicatie een noodzaak is kunnen als volgt ingedeeld worden.

(1) Wederzijdse uitsluiting

Van hulpbronnen en faciliteiten kan aangegeven worden dat ze *deelbaar* zijn, dat wil zeggen dat ze door gelijktijdige processen (dit in de betekenis zoals we in het voorafgaande besproken hebben) gebruikt kunnen worden. Of er kan aangegeven worden dat ze *ondeelbaar* zijn, wat inhoudt dat ze op één moment maar door één proces gebruikt kunnen worden. De ondeelbaarheid vloeit voort uit twee zaken:

(a) De fysieke aard van de hulpbron maakt deelgebruik onmogelijk. Een typisch voorbeeld hiervan is een ponsbandlezer, waarbij het ondoenlijk is tussen de letters door van band te wisselen.

(b) De hulpbron is van dien aard dat de acties van processen, die er gelijktijdig gebruik van maken, elkaar kunnen beïnvloeden. Een veel gebruikt voorbeeld is de geheugenplaats, met een variabele als inhoud, die voor meer dan één proces toegankelijk is: als het ene proces de waarde van de variabele bekijkt, terwijl het andere deze waarde verandert, dan zal het resultaat niet te voorspellen en waarschijnlijk rampzalig zijn. Stel je voor dat in een reserveringssysteem voor vliegtuigplaatsen de beschikbaarheid van een plaats weerspiegeld wordt door de inhoud van een bepaalde geheugenplaats. Als de geheugenplaats dan voor meer dan één proces tegelijk toegankelijk zou zijn, zou er

de mogelijkheid bestaan dat twee reisagenten dezelfde plaats boeken, door de hieronder beschreven ongelukkige volgorde van handelen te volgen.

Agent A ziet dat de plaats vrij is en overlegt met zijn klant

⋮

Agent A reserveert de plaats

Agent B ziet dat de plaats vrij is en overlegt met zijn klant

⋮

Agent B reserveert de plaats

Ondeelbare gebieden omvatten dan ook de meeste randapparatuur, bestanden waarin geschreven kan worden en gegevensgebieden die aan veranderingen onderhevig zijn. Deelbare hulpbronnen en faciliteiten zijn onder andere CVE's, bestanden die alleen gelezen kunnen worden en geheugengebieden die alleen maar procedures en gegevens bevatten die tegen verandering beschermd zijn.

Het probleem van wederzijdse uitsluiting is: het met zekerheid ervoor moeten zorgen dat niet deelbare hulpbronnen slechts voor één proces tegelijkertijd toegankelijk zijn.

(2) Synchronisatie

In het algemeen is de verhouding tussen de snelheden waarmee processen ten opzichte van elkaar verlopen onvoorspelbaar, omdat het afhangt van het aantal keren dat ze onderbroken worden en van de tijdverdeling van de processor tussen de processen. We zeggen dat processen *asynchroon* ten opzichte van elkaar verlopen. Om echter een goede samenwerking te krijgen zijn er bepaalde punten waar de processen hun activiteiten moeten gelijkschakelen. Dit zijn punten waarna een proces niet verder kan gaan voordat een ander proces klaar is met een bepaalde activiteit. Zo kan een proces dat een schema opstelt voor de gebruikersopdrachten niet verder voordat een inleesproces de laatste opdracht ingelezen heeft. Overeenkomstig mag een verwerkingsproces dat uitvoer tot gevolg heeft, niet in staat zijn verder te gaan voordat zijn vorige uitvoer afgedrukt is. Het behoort tot de verantwoordelijkheden van het besturingssysteem om mechanismen te verzorgen waarmee synchronisatie bereikt kan worden.

(3) Het 'vastlopen' van het systeem

Als een aantal processen tegelijkertijd hulpbronnen of faciliteiten probeert aan te spreken, is het mogelijk dat er een situatie ontstaat, waarin geen enkel proces verder kan, omdat de hulpbronnen die het nodig heeft door een ander proces in beslag genomen worden. Dit heet het *vastlopen* van het systeem (Engels: *deadlock* of *deadly embrace*). Het is te vergelijken met de verkeerschaos die ontstaat als twee naderende verkeersstromen elkaar proberen te kruisen en volkomen tot stilstand komen doordat ze elkaars weghelften blokkeren. Het vermijden van het vastlopen of het beperken van de gevolgen ervan is vanzelfsprekend een van de functies van een besturingssysteem.

3.3 SEINPALEN

De belangrijkste bijdrage aan de communicatie tussen processen was de introductie van het begrip *seinpalen* (Dijkstra 1965) en de simpele handelingen *passeer* en *verhoog* die erop werken. De termen *passeer* en *verhoog* zijn eveneens van Dijkstra afkomstig; in de Engelstalige literatuur gebruikt men vaak de termen *Wait* en *Signal*.

Een seinpaal (Engels: *semaphore*) is een niet negatief geheel getal waarmee men, afgezien van het de eerste keer aanroepen van zijn waarde, alleen kan werken met de handelingen *passeer* en *verhoog*. Deze handelingen werken alleen op seinpalen en het effect ervan is als volgt:

verhoog(s)

Het effect is dat de waarde van de seinpaal s met 1 toeneemt, de handeling van het toenemen wordt als ondeelbaar beschouwd. Deze ondeelbaarheid wil zeggen dat *verhoog(s)* niet gelijk is aan $s = s + 1$. Want stel dat twee processen A en B de handeling *verhoog(s)* willen uitvoeren als s bijvoorbeeld 3 is. De waarde van s zal dan 5 zijn als beide handelingen zijn voltooid. Stel echter dat in dezelfde omstandigheden beide processen de som $s = s + 1$ willen uitvoeren. Dan zal A de som $s + 1$ gaan maken en op 4 uitkomen; voordat A deze waarde aan s toekent, maakt B dezelfde som en komt op hetzelfde resultaat uit. De twee processen geven s dan de waarde 4, en een van de twee gewenste toenames is verloren gegaan.

passeer(s)

Het effect is het verminderen van de waarde van de seinpaal s met 1 zodat het resultaat niet negatief is. Ook hier is de handeling weer ondeelbaar. De *passeer*handeling houdt mogelijk een vertraging in, want als hij slaat op een seinpaal met de waarde 0, kan het proces, dat deze handeling uitvoert, pas verder op het moment dat een ander proces de waarde van de seinpaal naar 1 verhoogd heeft door het uitvoeren van een *verhoog*handeling. De ondeelbaarheid van de handeling houdt in dat, als meerdere processen opgehouden worden, er slechts één met goed gevolg zijn werk kan voortzetten op het moment dat de seinpaal positief wordt. Er wordt geen aanname gedaan over welk proces dit is.

De effecten van passeer- en verhooghandelingen kunnen als volgt samengevat worden:

passeer(s) : als $s > 0$ dan verminder s
verhoog(s) : verhoog s

waarbij s een seinpaal is.

Uit deze definitie kan men afleiden dat iedere *verhoog*handeling de waarde van een seinpaal met één verhoogt en dat iedere geslaagde (dat wil zeggen voltooide) *passeer*handeling de waarde met één vermindert. De waarde van een seinpaal is dus verbonden met het aantal *passeer*- en *verhoog*handelingen dat erop uitgevoerd is volgens de vergelijking

$$val(s) = C(s) + nv(s) - np(s) \qquad (3.1)$$

waar

$val(s)$ gelijk is aan de waarde van seinpaal
$C(s)$ gelijk is aan zijn beginwaarde
$nv(s)$ het aantal *verhoog*handelingen is dat erop uitgevoerd is
$np(s)$ het aantal geslaagde *passeer*handelingen is dat erop uitgevoerd is.

Maar per definitie is

$$val(s) \geq 0$$

Zo komen we tot de volgende belangrijke relatie

$$np(s) \leq nv(s) + C(s) \qquad (3.2)$$

waarbij het gelijkteken dan en slechts dan mogelijk is als $val(s) = 0$.

De relatie 3.2 verandert niet ten gevolge van *passeer*- en *verhoog*-handelingen; dat wil zeggen dat deze altijd geldt hoeveel handelingen er ook uitgevoerd worden.

Het implementeren van seinpalen met *passeer*- en *verhoog*handelingen wordt in het volgende hoofdstuk besproken. Voor het ogenblik zullen we maar zonder meer aannemen dat ze geïmplementeerd kunnen worden en zullen we ze gebruiken om eerder aangegeven problemen op te lossen.

(1) Wederzijdse uitsluiting

Ondeelbare hulpbronnen en faciliteiten, of dit nu randapparatuur, bestanden of gegevens in het geheugen zijn, kunnen beschermd worden tegen gelijktijdig gebruik door meerdere processen door middel van het blokkeren van die programmadelen, waarmee de toegang wordt verschaft. Deze programmadelen heten *kritische sectoren* en de wederzijdse uitsluiting van het brongebruik kan beschouwd worden als het wederzijds uitsluiten van het uitvoeren van kritische sectoren. Zo kunnen bijvoorbeeld processen elkaar de toegang beletten tot een gegevenstabel, als alle routines die de tabel lezen of bijwerken als kritische sectoren geschreven worden, zodat er maar één tegelijk uitgevoerd kan worden.

Het uitsluiten kan simpel gebeuren met behulp van het insluiten van elke kritische sector in *passeer*- en *verhoog*handelingen, die op een seinpaal met de beginwaarde 1 slaan. Iedere kritische sector wordt dus als volgt geprogrammeerd:

passeer (blokkeer);
kritische sector
verhoog (blokkeer);

waar *blokkeer* de naam van de seinpaal voorstelt.

De lezer zal, als hij dat controleert, zien dat als de beginwaarde van *blokkeer* 1 is, men inderdaad zeker is van wederzijdse uitsluiting, dit omdat er maar één proces is dat de handeling *passeer (blokkeer)* kan verrichten voordat een ander de handeling *verhoog (blokkeer)* verricht. Om precies te zijn resulteert de vergelijking 3.2 in:

$$np\,(blokkeer) \leq nv\,(blokkeer) + 1$$

Dit betekent dat er ten hoogste één proces tegelijk in de kritische sector kan zijn.

Verder wordt een proces nooit onnodig de toegang tot zijn kritische sector ontzegd; dit gebeurt alleen als een ander proces al in zijn kritische sector is. We kunnen ons dit realiseren als we bekijken dat een proces alleen de toegang ontzegd wordt als de waarde van *blokkeer* 0 is. In dat geval geeft de vergelijking 3.2 het volgende aan:

$$np\,(blokkeer) = nv\,(blokkeer) + 1$$

Met andere woorden, het aantal geslaagde *passeer*handelingen op *blokkeer* overtreft het aantal *verhoog*handelingen met 1, dat betekent dat er zich een proces in zijn kritische sector bevindt.

We concluderen hieruit dat, iedere keer dat een proces een, meervoudig gebruikte, variabele of hulpbron wil aanspreken, dit moet gebeuren door middel van een kritische sector die met een seinpaal, als hiervoor beschreven, beschermd moet worden.

Neem als voorbeeld een besturingssysteem dat twee processen bevat, *A* en *B*, die alle twee onderdelen aan een wachtrij toevoegen en verwijderen. Om ervoor te zorgen dat de wijzers (pointers) in de wachtrij niet in de war raken, kan het nodig zijn dat de toegang tot de wachtrij beperkt is tot één per keer. Het toevoegen en verwijderen van onderdelen aan de wachtrij zal met kritische sectoren gecodeerd moeten worden zoals het hieronder staat.

Programma voor proces A	*Programma voor proces B*
.	.
.	.
.	.
passeer (blokkeer)	*passeer (blokkeer)*
voeg onderdeel toe aan rij	verwijder onderdeel uit rij
verhoog (blokkeer)	*verhoog (blokkeer)*
.	.
.	.
.	.

Hier zou de oplettende lezer zich kunnen afvragen of de wederzijdse uitsluiting niet zonder het formele toevoegen van de seinpalen en de daarbij behorende handelingen opgelost had kunnen worden. Op het eerste gezicht kan dit vermoeden juist lijken, want het lijkt mogelijk een oplossing te bewerkstelligen door het beschermen van de kritische sectoren met een variabele (die we *poort* noemen; Engels: *gate*). Als de *poort open gezet* wordt (bijvoorbeeld aangegeven met de waarde 1), dan is de toegang tot de kritische sector toegestaan; als de *poort gesloten* is (aangegeven door een 0), dan is de toegang verboden. Een kritische sector zou dus als volgt gecodeerd worden:

```
als poort  = dicht dan geen actie
   poort := dicht

            kritische sector

   poort := open
```

Ieder proces dat de kritische sector ingaat test eerst of de *poort open* is (en blijft dat doen totdat de *poort open* is); en zet vervolgens de *poort dicht* om te voorkomen dat andere processen de sector ingaan.

Helaas werkt deze simpele oplossing niet. De reden hiervan ligt in de scheiding tussen het zoeken naar een open poort in regel één en het sluiten van de poort in regel twee. Ten gevolge van die scheiding kunnen twee tegelijkertijd verlopende processen op de eerste regel een open poort aantreffen voordat één van de twee de kans heeft die te sluiten. Het gevolg hiervan is dat beide processen de kritische sector samen binnengaan.

Seinpalen voorkomen dergelijke problemen door erop te staan dat de *passeer*- en *verhoog*handelingen ondeelbaar zijn; het is onmogelijk dat twee processen tegelijkertijd een handeling uitvoeren op dezelfde seinpaal. De implementatie van seinpalen, die in hoofdstuk 4 besproken zal worden, moet dat natuurlijk verzekeren.

(2) Synchronisatie

De eenvoudigste vorm van synchronisatie is dat proces *A* niet verder dan punt L1 mag gaan voordat een ander proces *B* punt L2 bereikt heeft. Voorbeelden van een dergelijke situatie zijn dat *A* op punt L1 informatie nodig heeft die *B* verzorgt als hij punt L2 bereikt. De synchronisatie kan als volgt geprogrammeerd worden:

Programma voor proces A	*Programma voor proces B*
.	.
.	.
.	.
L1: *passeer (doorgaan)*	L2: *verhoog (doorgaan)*
.	.
.	.
.	.

doorgaan is hier de seinpaal met beginwaarde 0.

Uit de bovenstaande programma's blijkt duidelijk dat *A* niet voorbij punt L1 kan voordat *B* de vervolghandeling bij punt L2 heeft uitgevoerd. (Het is natuurlijk zo dat als *B* die handeling uitvoert voordat *A* bij L1 aankomt er geen vertraging in *A* optreedt,) Opnieuw kunnen we formule 3.2 gebruiken om dat formeler te laten zien; we hebben

$$np\,(doorgaan) \leqq nv\,(doorgaan)$$

wat inhoudt dat *A* niet voorbij punt L1 kan voordat *B* voorbij L2 is.

Het bovenstaande proces is daarin asymmetrisch dat proces *A* door *B* geregeld wordt en niet omgekeerd. Het geval waarin twee processen elkaars voortgang regelen kunnen we met het klassieke voorbeeld van de producent en de afnemer laten zien. Omdat het typerend is voor veel problemen die zich voordoen bij de communicatie tussen processen zullen we er wat nader op ingaan.

Een aantal 'produktie'-processen en een aantal 'verbruiks'-processen communiceren met elkaar via een buffer die de producenten met onderdelen vullen, die er door de verbruikers uitgehaald worden. De producenten doorlopen steeds de cirkel 'maak onderdeel - stort het in buffer', de verbruikers doorlopen een soortgelijke cirkel 'haal onderdeel uit buffer - verbruik het'. Een typisch voorbeeld hiervan kan een verwerkingsproces zijn, dat regels uitvoer in een buffer plaatst, terwijl de bijbehorende verbruiker het proces kan zijn dat de regels afdrukt. De buffer heeft een beperkte capaciteit die groot genoeg is om N woorden van gelijke lengte te bevatten. De vereiste synchronisatie is tweeledig: ten eerste kunnen de producenten geen onderdelen in de buffer stoppen als die al vol is; en ten tweede kunnen de gebruikers geen onderdelen uit de buffer halen als die leeg is. Met andere woorden, als s het aantal gestorte onderdelen is en t het aantal is dat onttrokken werd, dan moeten we ervoor zorgen dat:

$$0 \leq s-t \leq N \qquad (3.3)$$

Verder moet de buffer beschermd worden tegen gelijktijdige benadering door meerdere processen, dit om te voorkomen dat de acties van de ene (bijvoorbeeld het bijwerken van de wijzers) door de andere verstoord worden. Zowel het storten als het onttrekken moeten dus als kritische sectoren gecodeerd worden.

We hebben dus de volgende oplossing voor ons probleem nodig:

Programma voor producenten

```
herhaal onbegrensd
begin
    produceer onderdeel
    passeer (ruimte vrij)
    passeer (bufferbesturing)
    stort onderdeel in buffer
    verhoog (bufferbesturing)
    verhoog (onderdeel
      beschikbaar)
einde
```

Programma voor verbruikers

```
herhaal onbegrensd
begin
    passeer (onderdeel
      beschikbaar)
    passeer (bufferbesturing)
    onttrek onderdeel uit
      buffer
    verhoog (bufferbesturing)
    verhoog (ruimte vrij)
    verbruik onderdeel
einde
```

De synchronisatie wordt met behulp van de seinpalen *ruimte vrij* en *onderdeel beschikbaar* bereikt, die als respectieve beginwaarden N en 0 hebben. De wederzijdse uitsluiting wordt bereikt met behulp van de seinpaal *bufferbesturing*, waarvan de beginwaarde 1 is.

We moeten nu laten zien dat deze oplossing voldoet aan vergelijking 3.3. Als we de invariante vergelijking 3.2 toepassen op de twee synchronisatie seinpalen, dan krijgen we:

$$np\,(\text{ruimte vrij}) \leq nv\,(\text{ruimte vrij}) + N \qquad 3.4$$

en

$$np\ (onderdeel\ beschikbaar) \leqq nv\ (onderdeel\ beschikbaar) \quad 3.5$$

Maar uit de volgorde van handelen, in het programma voor de produktie, leiden we af dat:

$$nv\ (onderdeel\ beschikbaar) \leqq s \leqq nv\ (ruimte\ vrij) \quad 3.6$$

en uit de volgorde van handelen, in het programma voor het verbruik, leiden we af dat:

$$nv\ (ruimte\ vrij) \leqq t \leqq np\ (onderdeel\ beschikbaar) \quad 3.7$$

Daarom komt uit 3.6

$$\begin{aligned} s &\leqq np\ (ruimte\ vrij) \\ &\leqq nv\ (ruimte\ vrij) && \text{uit 3.4} \\ &\leqq t + N && \text{uit 3.7} \end{aligned}$$

Net zo komt uit 3.7

$$\begin{aligned} t &\leqq np\ (onderdeel\ beschikbaar) && \text{uit 3.5} \\ &\leqq nv\ (onderdeel\ beschikbaar) && \text{uit 3.6} \\ &\leqq s \end{aligned}$$

Combinatie hiervan geeft

$$t \leqq s \leqq t + N$$

wat aantoont dat aan vergelijking 3.3 is voldaan.

De oplossing van het producenten-verbruikers probleem kan als leidraad gelden voor alle processen waartussen informatie wordt uitgewisseld. Om precies te zijn, processen die invoerapparatuur besturen, handelen als producenten voor die processen die de invoer gebruiken. De processen die uitvoerapparatuur besturen, handelen als verbruikers voor die processen die de uitvoer produceren.

(3) Het vastlopen van het systeem

We hebben er al eerder op gewezen dat het systeem kan vastlopen als meerdere processen toegang tot een hulpbron proberen te krijgen. In die hoedanigheid is het voorlopig een probleem dat we pas in hoofdstuk 8 zullen bespreken.

In dit deel merken we op dat het vastlopen ook kan optreden als gevolg van het feit dat processen wachten op het completeren van elkaars activiteiten. Neem bijvoorbeeld twee processen, *A* en *B*, die

handelingen uitvoeren op de sienpalen *X* en *Y* zoals hieronder.

```
proces A                    Proces B
   .                           .
   .                           .
   .                           .
passeer(X);                 passeer(Y);
   .                           .
   .                           .
   .                           .
passeer(Y);                 passeer(X);
   .                           .
   .                           .
   .                           .
```

Als de beginwaarden van *X* en *Y* 1 zijn, kan elk proces zijn eerste *passeer*handeling verrichten, en daarmee de waarden van de seinpalen tot 0 verlagen. Het zal duidelijk zijn dat geen van de twee processen voorbij zijn volgende *passeer*handeling komt, en het vastlopen is een feit.

Deze situatie komt in feite overeen met die waarin het vastlopen wordt veroorzaakt door competitie tussen aanvragen voor het gebruik van hulpbronnen. Als we een seinpaal beschouwen als een hulpbron en de handelingen *passeer* en *verhoog* als het aanvragen en vrijgeven hiervan, dan is het vastlopen ontstaan omdat zowel *A* als *B* de hulpbron blokkeren die de ander nodig heeft. De deelbaarheid van de seinpaal wordt bepaald door zijn beginwaarde: als de waarde *n* (> 1) is, dan kan de seinpaal door *n* processen gedeeld worden; als de waarde 1 is (zoals hierboven) dan is de seinpaal ondeelbaar.

Een vergelijkbare maar complexere vorm van vastlopen kan afgeleid worden uit het producent-verbruiker probleem, en wel door de volgorde van de *passeer*handelingen, die door de verbruikers worden uitgevoerd, om te draaien. Als de buffer leeg is, kan de verbruiker toegang krijgen tot de buffer door het uitvoeren van de handeling *'passeer (bufferbesturing)'* waarna het blijft hangen op *'verhoog (onderdeel beschikbaar)'*. De enige manier om de verbruiker weer vrij te krijgen is het storten van een onderdeel in de buffer door een producent, samen met het geven van *'verhoog (onderdeel beschikbaar)'*. Er is echter geen producent die toegang tot de buffer kan krijgen omdat die door de geblokkeerde verbruiker bezet wordt. Het gevolg is dus vastlopen. Ook hier is de situatie weer te vergelijken met die waarin er 'gevochten' wordt om de toegang tot hulpbronnen. In dit geval zijn de hulpbronnen de buffer, die door de verbruiker geblokkeerd wordt, en het nieuwe onderdeel, dat door een producent vastgehouden wordt.

De les die uit deze voorbeelden geleerd kan worden is dat het vastlopen kan optreden als gevolg van een verkeerd opgestelde volgorde van *passeer*handelingen, zelfs wanneer er geen expliciete

verzoeken voor toegang worden gedaan. Het kan gezond zijn je te realiseren dat het risico van vastlopen, als gevolg van een verkeerde oplossing van het producent-verbruiker vraagstuk, niet opgemerkt zou worden door het uitvoeren van een controle op juistheid zoals in de vorige paragraaf is besproken.

Zowel de oplossing met, als die zonder het risico van vastlopen voldoet aan vergelijking 3.3 en zijn wat dat criterium betreft dus even goed. In hoofdstuk 8 zullen we criteria bespreken voor het herkennen van vastlopen met betrekking tot de toewijzing van hulpbronnen; het herkennen van het al dan niet vastlopen in een opeenvolgende serie *passeer*handelingen kan met een analyse gebeuren zoals die hierna beschreven wordt.

Laten we nog eens kijken naar de oplossing van het producent-verbruiker probleem, die we in sectie 2 gaven. We zullen aantonen dat die inderdaad niet vastloopt. Eerst merken we op dat de binnenste delen van de programma's voor de producenten en voor de verbruikers normale kritische sectoren zijn die door de seinpaal *buffer-besturing* beveiligd zijn; daarbinnen doet zich dus geen gevaar voor vastlopen voor. Hieruit volgt dan dat vastlopen alleen maar kan voorkomen als

(a) geen producent voorbij *passeer (ruimte vrij)* kan, en geen verbruiker *verhoog (ruimte vrij)* kan uitvoeren (dat wil zeggen dat geen verbruiker onderdelen onttrekt)

en als

(b) geen verbruiker voorbij *passeer (onderdeel beschikbaar)* kan, en geen producent *verhoog (onderdeel beschikbaar)* kan uitvoeren (dat wil zeggen dat geen producent stort).

Voorwaarde (a) samen met vergelijking 3.2 betekent dat:

$$np\ (ruimte\ vrij) = nv\ (ruimte\ vrij) + N$$

en dat

$$np\ (onderdeel\ beschikbaar) = nv\ (ruimte\ vrij)$$

Voorwaarde (b) samen met vergelijking 3.2 betekent dat:

$$np\ (onderdeel\ beschikbaar) = nv\ (onderdeel\ beschikbaar)$$

en dat

$$np\ (ruimte\ vrij) = nv\ (onderdeel\ beschikbaar)$$

Als we deze vergelijkingen combineren krijgen we:

np (ruimte vrij) = np (ruimte vrij) + N

wat onmogelijk is omdat $N > 0$.

Een gelijksoortige analyse kan niet gemaakt worden in het geval dat de volgorde van de *passeer*handelingen in het verbruikersprogramma niet langer een normale kritische sector is (het bevat nu een *passeer*-handeling). Er is een complexere analyse voor nodig om aan te tonen dat het vastlopen inderdaad kan voorkomen (bijvoorbeeld Habermann, 1972; Lister, 1974).

3.4 BEWAKINGSPROGRAMMA'S (MONITORS)

In het vorige deel zagen we hoe seinpalen voor het synchroniseren van en het communiceren tussen processen gebruikt kunnen worden. Bij ongedisciplineerd gebruik van seinpalen is het risico van fouten echter nogal groot - een programmeur kan gemakkelijk *passeer*- en *verhoog*handelingen op de verkeerde plaatsen zetten, of ze zelfs helemaal niet zetten. Als een programmeur zich bijvoorbeeld niet realiseert dat een bepaalde gegevensstructuur door meerdere processen gebruikt wordt, dan zal hij nalaten de kritische sectoren, die de toegang tot de gegevensstructuur verzorgen, tussen *passeer*- en *verhoog*handelingen in te zetten. De gegevensstructuur is dan onbeschermd tegen gelijktijdige bewerkingen door verschillende processen, en onverenigbaarheden in de inhoud zullen er waarschijnlijk het gevolg van zijn.

Om dit te voorkomen is er een aantal voorstellen geweest voor constructies van programmeertalen die de programmeur verplichten tot het expliciet aangeven van de deelbaarheid van hulpbronnen en die wederzijdse uitsluiting voor dergelijke objecten verplicht stellen. Een van de invloedrijkste en meest aanvaarde constructies is de *monitor* (Hoare, 1974), die hieronder kort en in de appendix uitgebreider besproken wordt.

Een monitor bestaat uit:

(1) de gegevens voor een gemeenschappelijk object;
(2) een stel procedures die aangeroepen kunnen worden voor het verkrijgen van toegang tot het object;
(3) een programmadeel dat het object initialiseert (dit programma wordt maar één keer, bij het creëren van het object, uitgevoerd).

Een buffer (zoals in het vorige deel besproken) voor het doorgeven van gegevens tussen producent en verbruiker kan met behulp van een monitor als volgt worden weergegeven:

(1) de bufferruimte en 'wijzers' (bijvoorbeeld een reeks gegevens met de daarbij behorende indexen);

(2) twee procedures *storten* en *onttrekken* die door processen kunnen worden aangeroepen voor het uitvoeren van de respectieve acties op onderdelen;
(3) een programmadeel dat de wijzers van de buffer, bij het initialiseren, naar het begin van de buffer zet.

Een vertaalprogramma voor een computertaal die monitoren bevat moet ervoor zorgen dat de toegang tot een gedeeld object alleen verkregen kan worden door middel van het aanroepen van de erbij behorende monitorprocedure (dit kan vrij gemakkelijk bereikt worden door het bewakingsmechanisme dat zich in de meeste talen met een blokstructuur bevindt). Het vertaalprogramma moet ook verzekeren dat de procedures van de diverse monitoren als kritische sectoren geïmplementeerd zijn die elkaar uitsluiten. Dus garandeert het vertaalprogramma, in het geval van de buffer, dat de toegang daartoe voorbehouden wordt aan de *stort-* en *onttrek*procedures en dat deze elkaar wederzijds uitsluiten.

Het zal duidelijk zijn dat monitoren een mogelijk vruchtbare bron van fouten uitschakelen door de verantwoordelijkheid voor de wederzijdse uitsluiting bij het vertaalprogramma te leggen in plaats van bij de programmeur. De verantwoordelijkheid voor andere vormen van synchronisatie blijft echter bij de programmeur liggen, die daar seinpalen (of iets dergelijks) voor moet gebruiken. De weergave van een buffer in de vorm van een monitor (zoals hierboven) verzekert, door het wederzijds uitsluiten van *storten* en *onttrekken*, dat er geen onderdelen tegelijkertijd in de buffer gestort of aan de buffer onttrokken kunnen worden. Het voorkomt echter niet dat er onderdelen in een volle buffer gestort worden of aan een lege buffer onttrokken worden. Dergelijke rampen moeten voorkomen worden door het plaatsen van daarvoor geschikte synchronisatiehandelingen in de *stort-* en *onttrek*procedures. De details worden in de Appendix gegeven.

De monitorconstructie is in een aantal programmeertalen geïmplementeerd zoals: Concurrent Pascal (Hansen, 1975), Pascal-plus (Welsh en Bustard, 1979) en Mesa (Lampson en Redell, 1980). Voor degene die het besturingssysteem moet implementeren heeft dat het voordeel dat alle handelingen op een gedeeld object beperkt worden tot het hanteren van een stel goed gedefinieerde procedures. En het heeft het voordeel dat hij ervan verzekerd is dat deze procedures elkaar wederzijds uitsluiten. Synchronisaties met een ander oogmerk dan de wederzijdse uitsluiting blijven de verantwoordelijkheid van de programmeur.

3.5 SAMENVATTING

In dit hoofdstuk hebben we het begrip 'proces' geïntroduceerd en dat tegenover 'programma' gesteld. We hebben ook laten zien dat het een handig begrip kan zijn bij de bestudering van de karakteristieken van een besturingssysteem. We hebben in het bijzonder gezien dat 'gelijktijdigheid' en 'flexibele respons' beschreven kunnen worden in termen van het overschakelen van processoren tussen processen. We hebben ook 'seinpalen' met de daarbij behorende handelingen geïntroduceerd als een mechanisme voor de communicatie tussen de procesen onderling en we hebben aangetoond dat ze als gereedschap voldoen voor het oplossen van de problemen bij de wederzijdse uitsluiting en de synchronisatie. Tenslotte hebben we gezien dat het in sommige gevallen mogelijk is bepaalde vergelijkingen voor de waarden van seinpalen te gebruiken, om aan te tonen dat processen zich gedragen zoals we dat verwachten.

In het volgende hoofdstuk zullen we deze gereedschappen gaan gebruiken voor het construeren van ons 'papieren' besturingssysteem.

4 De systeemkern

In het vorige hoofdstuk hebben we de begrippen en gereedschappen ontwikkeld die nodig zijn voor het maken van het in hoofdstuk 1 geschetste besturingssysteem. Zoals we daar al hebben vermeld, zal het papieren besturingssysteem in opbouw lijken op een ui, waarbij elke laag een set functies voorstelt die alleen van de erbinnen liggende lagen afhankelijk is. In het midden van de ui zetelen de faciliteiten waarin door de hardware van de machine zelf wordt voorzien. De lagen kunnen beschouwd worden als het aanbrengen van opeenvolgende virtuele machines, zodat de gehele ui de virtuele machine weergeeft die de gebruiker nodig heeft.

De belangrijkste koppeling tussen de kale hardware en het besturingssysteem wordt verzorgd door de *systeemkern*, dit is de binnenste laag van de ui. Het doel van de kern is het zorgen voor een omgeving waarin processen kunnen bestaan. Dit houdt de afhandeling van interrupts in, het schakelen van processoren tussen processen en het implementeren van mechanismen voor de communicatie tussen de processen. Voordat we deze functies nader gaan beschrijven zullen we een blik werpen op de essentiële hardware die nodig is voor de ondersteuning van het besturingssysteem dat we proberen op te bouwen.

4.1 NOODZAKELIJKE HARDWAREFACILITEITEN

(1) Het interruptmechanisme

In hoofdstuk 2 werd opgemerkt dat het voor het laten overlappen van I/O activiteiten met de centrale verwerking nodig is dat het lopende proces onderbroken kan worden, als het overbrengen van gegevens naar randapparatuur voltooid is. Het is daarom noodzakelijk dat onze computer voorziet in een *interruptmechanisme* (of *ingreepmechanisme*) dat op zijn minst de waarde van de programmateller bewaart en de besturing overdraagt naar een vaste plaats in het geheugen. Deze geheugenplaats zal gebruikt worden als het begin van een programmastuk dat *interruptbesturing* heet (Engels: *interrupt handler* of *interrupt routine*).

Het doel van die routine is het vaststellen van de bron van de interrupt en het hierop reageren op de juiste manier. We zullen in sectie 4.4 de ingreeproutine beschrijven en we zullen de diverse vormen bespreken die deze kan aannemen, dit al naar gelang van de preciese aard van het beschikbare interruptmechanisme.

(Volledigheidshalve vermelden we dat enkele computers, bijvoorbeeld de CDC CYBER 170 serie, werken zonder een expliciet interruptmechanisme. In een dergelijke computer moeten één of meerdere processoren bestemd zijn voor het bewaken van de toestand van de I/O apparatuur om te signaleren of overdrachten voltooid zijn. In het geval van de CYBER wordt deze functie door een aantal zogenaamde 'randapparatuurprocessoren' uitgevoerd, terwijl de CVE van alle I/O besturing verlost wordt. Er bestaat echter nog een beperkte vorm van interrupts, omdat randapparatuurprocessoren de CVE kunnen dwingen tot het springen naar een andere lokatie.)

(2) Geheugenbescherming

Als er meerdere processen tegelijkertijd actief zijn is het nodig de geheugenruimte die door het ene proces gebruikt wordt te beschermen tegen niet toegestane benadering van dat geheugen door een ander proces. De beschermingsmechanismen die in de hardware van de geheugenadressering ingebouwd moeten worden zullen in het volgende hoofdstuk nader beschreven worden; voorlopig nemen we het bestaan ervan zonder meer aan.

(3) Een bevoorrechte instructieset

Om ervoor te zorgen dat gelijktijdig lopende processen elkaar niet kunnen beïnvloeden, moet een deel van de instructieset van de computer gereserveerd worden voor het exclusieve gebruik van het besturingssysteem. Deze bevoorrechte instructies voeren functies uit zoals:

(a) het mogelijk en onmogelijk maken van ingrepen,
(b) het schakelen van een processor tussen processen,
(c) de toegang tot registers, gebruikt door de hardware voor de geheugenbescherming,
(d) het verzorgen van in- of uitvoer,
(e) het stoppen van een centrale processor.

Om onderscheid te maken tussen de keren dat bevoorrechte instructies wel of niet zijn toegestaan, werken de meeste computers in een van de twee volgende toestanden die meestal de *supervisortoestand* en de *gebruikerstoestand* genoemd worden. De overgang van de gebruikerstoestand in de supervisortoestand vindt automatisch plaats in een van de volgende omstandigheden.

(a) Een gebruikersproces verzoekt het besturingssysteem een systeemfunctie uit te voeren waarbij het gebruik van een bevoorrechte instructie nodig is. Een dergelijk verzoek heet *extracode* of *supervisoroproep*.
(b) Als er zich een interrupt voordoet.
(c) Als er zich een foutsituatie voordoet in een gebruikersproces. De situatie kan als 'interne interrupt' beschouwd worden en in eerste instantie door een interruptroutine behandeld worden.
(d) Er wordt een poging ondernomen tot het uitvoeren van een bevoorrechte instructie, terwijl de machine in de gebruikerstoestand is. De poging kan beschouwd worden als een bepaald soort fout en kan zoals in (c) hierboven afgehandeld worden

De terugschakeling van de supervisortoestand naar de gebruikerstoestand wordt bewerkstelligd door een instructie die zelf bevoorrecht is.

Het is misschien de moeite waard op te merken dat het bovengenoemde schema met twee voorrangsniveaus in sommige computers uitgebreid is naar meerdere niveaus (vijvoorbeeld de ICL 2900, de CYBER 180 en de Honeywell 645). Tot hoofdstuk 9 zullen wij afzien van verdere bespreking van dergelijke machines met meerdere niveaus.

(4) Real-time klok

Een hardwareklok die op vaste intervallen 'real-time' onderbreekt (dat wil zeggen dat de tijd gemeten wordt ten opzichte van de buitenwereld en niet ten opzichte van de tijd die voor een verwerking door een bepaald proces nodig is) is onontbeerlijk bij de implementatie van indelingsstrategieën en voor het verantwoorden van hulpbronnen die door de verschillende gebruikers gebruikt worden.

Van nu af zullen wij aannemen dat de machine, waarop wij ons papieren besturingssysteem bouwen, de bovengenoemde hardwarefaciliteiten heeft. Om precies te zijn, omdat we de mogelijkheid van het gebruik van meerdere processoren in een configuratie openhouden, zullen we aannemen dat iedere processor de faciliteiten (1) t/m (3) heeft en dat er een enkele real-time klok is die alle processoren kan onderbreken. Daarbij zullen wij ons verder beperken tot *hecht gekoppelde* configuraties waarin de processoren identiek zijn en samen een geheugen delen. Dit sluit de bespreking van computernetwerken uit: processoren hebben daar hun eigen geheugen en er wordt gecommuniceerd met behulp van datacommunicatie. Ook sluit dat systemen uit die zo georganiseerd zijn dat de werklast onder, onderling verschillende, processoren zo verdeeld wordt dat er optimaal gebruik gemaakt wordt van hun karakteristieken. Voor het eerste onderwerp zou op zich al een heel boek nodig zijn; het tweede is nog een onontgonnen gebied van onderzoek waaraan nog geen duidelijke richting is gegeven.

4.2 SCHETS VAN DE SYSTEEMKERN

De relatie tussen de systeemkern en de rest van het besturingssysteem is in figuur 4.1 geïllustreerd. De horizontale bodemlijn in het diagram geeft de computerhardware aan; het dikkere deel van die lijn is de set bevoorrechte instructies waarvan we het gebruik beperken tot de kern. (Uitzonderingen op die regel zijn enkele bevoorrechte instructies die te maken hebben met geheugenbescherming en I/O zoals we in volgende hoofdstukken zullen bespreken.)
De kern bestaat uit drie programmadelen:

(1) de *Basis-Niveau Ingreep Besturing*, dat de eerste afhandeling van alle interrupts verzorgt;
(2) het *verdeelprogramma* (Engels: *dispatcher*), dat de centrale processor schakelt tussen de processen;
(3) twee procedures (routines) die de basiscommando's voor communicatie tussen processen, *passeer* en *verhoog* (beschreven in hoofdstuk 3), implementeren. Deze procedures kunnen in de betrokken processen door middel van extracodes aangeroepen worden.

Omdat de kern rechtstreeks op de kale hardware gebouwd is, kunnen we verwachten dat dit het meest machine-afhankelijke deel van een besturingssysteem is. Het is waarschijnlijk zelfs het enige deel van het besturingssysteem dat voornamelijk in assembleertaal geschreven is: op een paar uitzonderingen na, die voornamelijk met I/O te maken hebben, kunnen de andere lagen zonder problemen in een hogere programmeertaal gecodeerd worden. Bruikbare programmeertalen voor systemen, die interessant kunnen zijn voor de lezer, zijn BCPL (Richards, 1969), BLISS (Wulf c.s., 1971) en C (Kernighan and Ritchie, 1978). Andere talen, zoals Concurrent Pascal (Hansen, 1975) en Modula (Wirth, 1977 en 1983), zijn speciaal bedoeld voor het schrijven van besturingssystemen, alhoewel zij enige eisen stellen aan de systeemontwerper. De beperking van het gebruik van assembleertaal tot een klein deel van het besturingssysteem (niet meer dan 400 instructies) die mogelijk is door de

Figuur 4.1 *Structuur van het papieren besturingssysteem*

hiërarchische structuur van het systeem, kan in grote mate bijdragen tot het realiseren van het doel: een foutloos, te begrijpen en te onderhouden produkt.

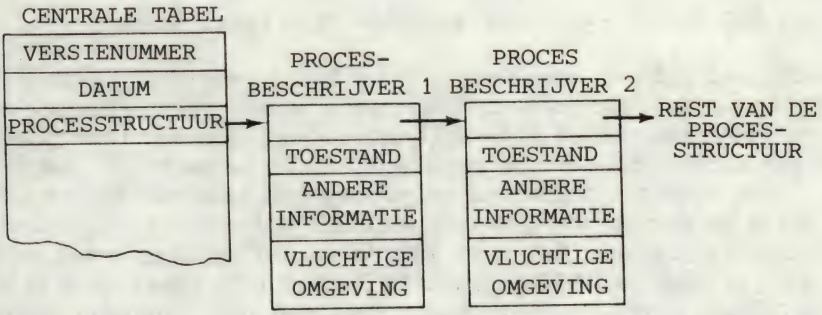
4.3 DE WEERGAVE VAN PROCESSEN

De programma's in de kern zijn verantwoordelijk voor het handhaven van een omgeving waarin processen kunnen bestaan. Dientengevolge wordt ervan verwacht dat zij werken met een soort gegevensstructuur die de fysische weergave is van alle processen in het systeem. Dit deel gaat over de aard van deze gegevensstructuur.

Elk proces kan worden weergegeven door een *procesbeschrijver* (soms ook wel *control block* of *state vector* genaam); dit is een gedeelte van het geheugen dat alle relevante informatie over het proces bevat. De samenstelling van deze informatie wordt duidelijk als we verder gaan; voorlopig laten we het hier een soort identificatie van het proces (de *procesnaam*) en informatie over de *toestand* van het proces omvatten. Een proces kan zich in principe in drie toestanden bevinden: het *wordt uitgevoerd*: wat inhoudt dat het door een processor uitgevoerd wordt; het kan *uitvoerbaar* zijn, wat betekent dat het zou kunnen lopen als er een processor aan toegewezen zou worden; of het is *onuitvoerbaar*, wat betekent dat het geen gebruik kan maken van een processor, zelfs als er een processor aan toegewezen zou worden. De meest voorkomende reden voor het onuitvoerbaar zijn van een proces is dat het wacht op het voltooien van gegevensoverdracht naar randapparatuur. De toestand van een proces is een essentieel stuk informatie voor het verdeelprogramma bij het toewijzen van een centrale processor aan processen.

We zullen aan een proces dat op een centrale processor loopt refereren met de term *huidig proces* voor die processor; het aantal huidige processen is natuurlijk kleiner dan of gelijk aan het aantal beschikbare processoren.

Een volgend deel van de procesbeschrijver kan gebruikt worden voor het opslaan van alle informatie over het proces die bewaard moet worden als de processor ophoudt met de besturing van het proces. Deze informatie, die nodig is voor de latere hervatting van het proces, omvat de waarden van alle machineregisters zoals: programmatellers, registers voor rekenkundige bewerkingen en indexregisters, die door een ander proces beïnvloed zouden kunnen worden. Het omvat ook de waarden van alle registers die gebruikt worden bij het aangeven van het geheugen dat aan een proces toebehoort (zie ook hoofdstuk 5). We noemen dit geheel de *vluchtige omgeving* van het proces (andere in de literatuur voorkomende namen zijn *context block* en *proces state*). De vluchtige omgeving van een



Figuur 4.2 *Centrale tabel en processtructuur*

proces kan formeler gedefinieerd worden als dat deel van de veranderbare, gemeenschappelijk bruikbare faciliteiten van het systeem die voor een proces toegankelijk zijn.

De procesbeschrijver van ieder proces is gekoppeld aan een *processtructuur*, die dienst doet als een beschrijving van alle processen in het systeem. Voor het moment zullen we een elementaire vorm van processtructuur aannemen, waarin de procesbeschrijvers gekoppeld zijn aan een eenvoudige lijst. De processtructuur is de eerste gegevensstructuur die we in ons besturingssysteem ingevoerd hebben; omdat het bij lange na de laatste niet zal zijn, voeren we ook een *centrale tabel* in, waarvan het doel is het verzorgen van toegang tot alle systeemstructuren. De centrale tabel zal een wijzer (Engels: pointer) bevatten naar elke datastructuur en kan ook gebruikt worden voor het opslaan van algemene gegevens zoals: datum, tijd van de dag, en versienummer van het systeem, dat van tijd tot tijd nodig kan zijn. De centrale tabel en de gegevensstructuur zijn in figuur 4.2 weergegeven.

4.4 DE BASIS-NIVEAU INGREEP BESTURING (BNIB)

De Basis-Niveau Ingreep Besturing (BNIB) is dat deel van het besturingssysteem dat verantwoordelijk is voor de behandeling van signalen uit de buitenwereld (interrupts) en vanuit het computersysteem zelf (foutsituaties en extracodes). We zullen aan beide types refereren met de term interrupts of ingrepen, en we zullen de bijwoorden 'extern' en 'intern' gebruiken om, daar waar nodig, onderscheid te maken. De functie van de BNIB is tweevoudig:

(1) het vaststellen van de bron van de ingreep,

(2) het aanroepen van de betreffende interruptroutine.

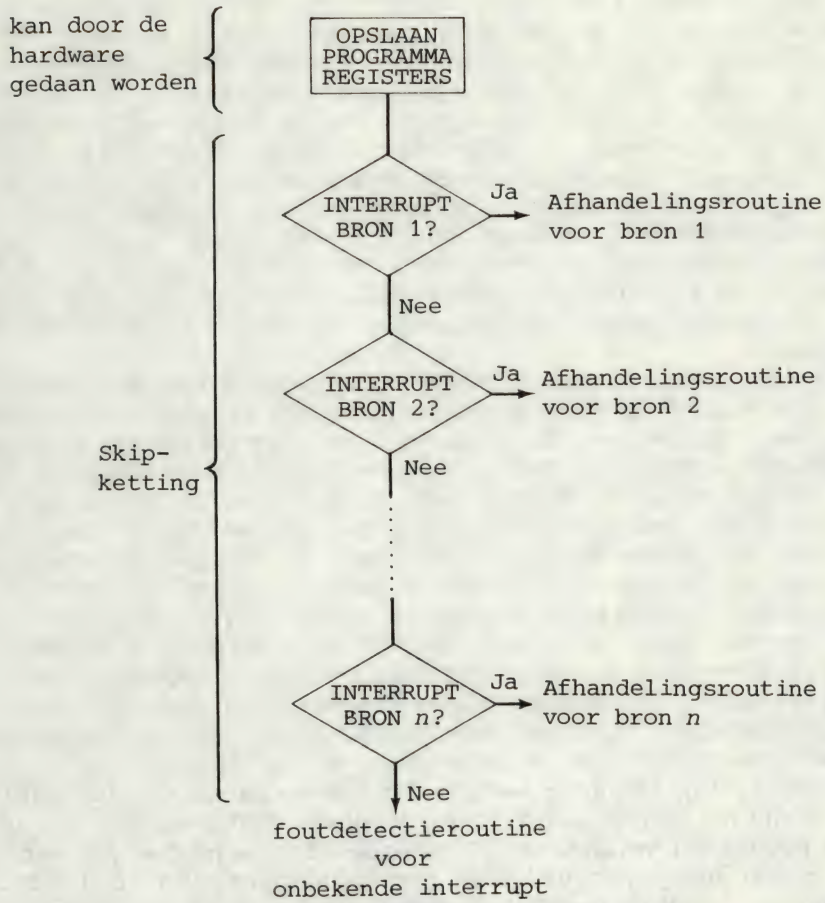
In sectie 4.1 hebben we al gesteld dat het ingreepmechanisme van de computer verantwoordelijk is voor het bewaren van de laatste waarde van de programmateller van het onderbroken proces. Ook moet ervoor gezorgd worden dat andere registers, die door de BNIB vereist worden en die door het onderbroken proces gebruikt worden, op een geschikte manier opgeslagen worden. Als dat niet gebeurt door het interruptmechanisme, dan moet het de eerste handeling van de BNIB zelf zijn. Omdat de BNIB een relatief eenvoudig programma is dat in een specifiek daarvoor aangewezen deel van het geheugen werkt, zal het aantal hierbij betrokken registers niet groot zijn, misschien maar een enkel register in de CVE. Het zal in elk geval aanzienlijk kleiner zijn dan de vluchtige omgeving van het onderbroken proces, dat niet in zijn geheel opgeslagen hoeft te worden omdat het hervat kan worden als de interrupt afgehandeld is.

Een alternatieve strategie voor het opslaan van registerwaarden, die bijvoorbeeld in de Z80 van Zilog, de IBM serie I en enkele machines van de PDP-11 groep toegepast wordt, is het voorzien in een extra set registers die alleen in de supervisortoestand gebruikt wordt. De BNIB kan hiervan gebruik maken en die van het onderbroken proces intact laten.

Het vaststellen van de oorzaak van de interrupt kan, afhankelijk van de aanwezige hardware, relatief gemakkelijk gebeuren. Bij de allerprimitiefste hardware, waarbij elke interrupt de besturing naar dezelfde plaats overdraagt, moet de identificatie geschieden door één serie testen op de statusvlaggen (tekens die een toestand aangeven) van alle mogelijk bronnen. Deze serie, die vaak *skipketting* (Engels: *skip chain*) genoemd wordt, is in Figuur 4.3 weergegeven. Het heeft duidelijk voordelen de skipketting zo te coderen dat de meest voorkomende bronnen van interrupts vooraan staan.

Bij sommige computers (bijvoorbeeld de PDP-11) is de skipketting overbodig door het gebruik van een stukje hardware dat onderscheid maakt tussen de interruptbronnen, door voor elke bron de besturing naar een andere plaats over te dragen. Dit vermindert de tijd die nodig is voor het identificeren van een interrupt, maar dat kost extra geheugenplaatsen voor de interrupts. Een tussenoplossing die bij een aantal computers, inclusief de IBM 370 serie, toegepast wordt, is het voorzien in een klein aantal interruptgeheugenplaatsen die elk door een groep randapparaten gebruikt worden. Het eerste deel van de identificatie van een interrupt wordt dan verzogd door de hardware. Een korte skipketting, die bij elke geheugenplaats begint, is voldoende om de identificatie af te maken. Het onderscheid tussen externe interrupts, foutsituaties en extracodes wordt vaak op die manier gemaakt. Het interruptmechanisme, zoals op de IBM 370, kan verder helpen bij de identificatie, doordat informatie over de interrupt in een vaste geheugenplaats gezet wordt.

Interrupts op de centrale processor worden normaliter onderdrukt als deze overschakelt van de gebruikers- naar de supervisortoestand. Dit zorgt ervoor dat de waarden van registers, die bij het



Figuur 4.3 *Interrupt identificatie met behulp van een skipketting*

ingaan van de BNIB opgeslagen werden, overschreven worden door een daaropvolgende interrupt die optreedt als de BNIB verlaten wordt. Een interrupt die voorkomt als het interruptmechanisme buiten werking is gesteld, wordt 'hangende' gehouden tot het mechanisme weer in werking gesteld wordt, bij het weer ingaan van de gebruikerstoestand. Deze regeling wordt onwerkbaar als er een groot verschil bestaat in de benodigde responstijd van randapparatuur, als er bijvoorbeeld geen gegevens verloren mogen gaan. In deze gevallen is het geven van *prioriteiten* aan de diverse interruptbronnen gemakkelijk, net zo als het gemakkelijk is de mogelijkheid te geven een interruptroutine zelf te onderbreken als een apparaat van hogere prioriteit om diensten vraagt. Sommige computers (bijvoorbeeld de IBM 370) geven de mogelijkheid hiertoe door het selectief onmogelijk maken van interrupts in de BNIB; als de BNIB bezig

is met een interrupt, worden alle interrupts met gelijke of lagere prioriteit door de BNIB onmogelijk gemaakt. Er moet uiteraard voor gezorgd worden dat de programmaregisters van het onderbroken proces in andere geheugenplaatsen opgeslagen worden, dit in overeenstemming met het prioriteitsniveau van de ontvangen interrupt. Een andere mogelijkheid is, dat de interrupthardware zelf onderscheid kan maken tussen de diverse prioriteitsniveaus, de besturing kan overgeven en de registers op kan slaan op verschillende geheugenplaatsen voor ieder niveau. Een interrupt op een bepaald niveau blokkeert automatisch andere op hetzelfde of lagere niveaus. Het DEC System-10, de PDP-11 en de M6800 zijn voorbeelden van processoren met een dergelijk interruptmechanisme, zij hebben alle acht prioriteitsniveaus.

De tweede functie van de BNIB is het starten van de afhandeling van een interrupt door het aanroepen van een geschikte afhandelingsroutine voor interrupts, die passend is voor de betrokken interrupt. In hoofdstuk 6 zullen we de details geven voor afhandelingsroutines voor I/O apparatuur en in hoofdstuk 11 zullen we die geven voor de afhandeling van foutdetectieroutines. Op dit ogenblik merken we slechts op dat, omdat de interruptroutines in de supervisortoestand lopen, waarbij de ingrepen geheel of gedeeltelijk geblokkeerd zijn, het wenselijk is ze zo kort mogelijk te houden. In het algemeen zal iedere routine de een of andere minimale actie uitvoeren, bijvoorbeeld het doorgeven van een letterteken van een invoerapparaat naar een buffer, en het overdragen van de verantwoordelijkheid voor verdere actie zoals het reageren op het ontvangen letterteken, naar een proces dat in de normale gebruikerstoestand loopt.

Het is belangrijk op te merken dat het voorkomen van een interrupt, of die nu extern of intern is, vaak de status van het een of andere proces zal veranderen. Bijvoorbeeld, een proces dat verzocht heeft om een gegevensoverdracht naar randapparatuur en dat onuitvoerbaar is terwijl de overdracht plaatsvindt, wordt weer uitvoerbaar gemaakt door de interrupt die plaatsvindt bij de voltooiing van de overdracht. Ook hier weer zien we dat bepaalde extracodes, zoals een *passeer*handeling op een seinpaal met de waarde nul, of een verzoek voor I/O, tot resultaat zal hebben dat het huidige proces niet verder kan. In alle gevallen wordt de verandering van de status veroorzaakt door de interruptroutine die de statusingang van de procesbeschrijver van het betrokken proces verandert.

Een gevolg van de statusverandering is, dat het proces dat op de betrokken processor liep voordat de interrupt plaatsvond, niet het meest passende kan zijn om daarna op die processor te gaan lopen. Het kan bijvoorbeeld het geval zijn dat de interrupt een proces met een hogere prioriteit dan het huidige uitvoerbaar maakt. De vraag wanneer een processor tussen processen moet overschakelen en ten gunste van welk proces, wordt in de volgende sectie behandeld.

4.5 HET VERDEELPROGRAMMA

Het is de taak van het *verdeelprogramma* (Engels: *dispatcher* of ook wel *low-level scheduler*) om de toewijzing te verzorgen van de centrale processoren aan de diverse processen in het systeem. Het verdeelprogramma wordt altijd aangeroepen als een lopend proces niet verder kan, of als er gronden zijn om aan te nemen dat een processor ergens anders beter ingezet kan worden. Deze gevallen kunnen als volgt beschreven worden:

(1) na een externe interrupt die de status van een proces verandert;
(2) na een extracode die tot gevolg heeft dat een proces tijdelijk niet verder voortgezet kan worden;
(3) na een foutdetectie die een tijdelijke onderbreking van het huidige proces tot gevolg heeft, totdat die fout afgehandeld is.

Dit zijn allemaal speciale gevallen van interrupts; dat wil zeggen het zijn allemaal interrupts die de status van het een of andere proces veranderen. Ten behoeve van de eenvoud onderscheiden we deze niet van die interrupts die de status van een proces niet veranderen, zoals een *passeer*handeling op een seinpaal met een positieve waarde; we zeggen in feite dat het verdeelprogramma ten slotte na *alle* interrupts actief wordt. Het extra werk, veroorzaakt door het ingaan van het verdeelprogramma bij gevallen waarbij er geen status is veranderd, wordt meer dan goedgemaakt door de voordelen die voortvloeien uit een uniforme behandeling van alle interrupts.

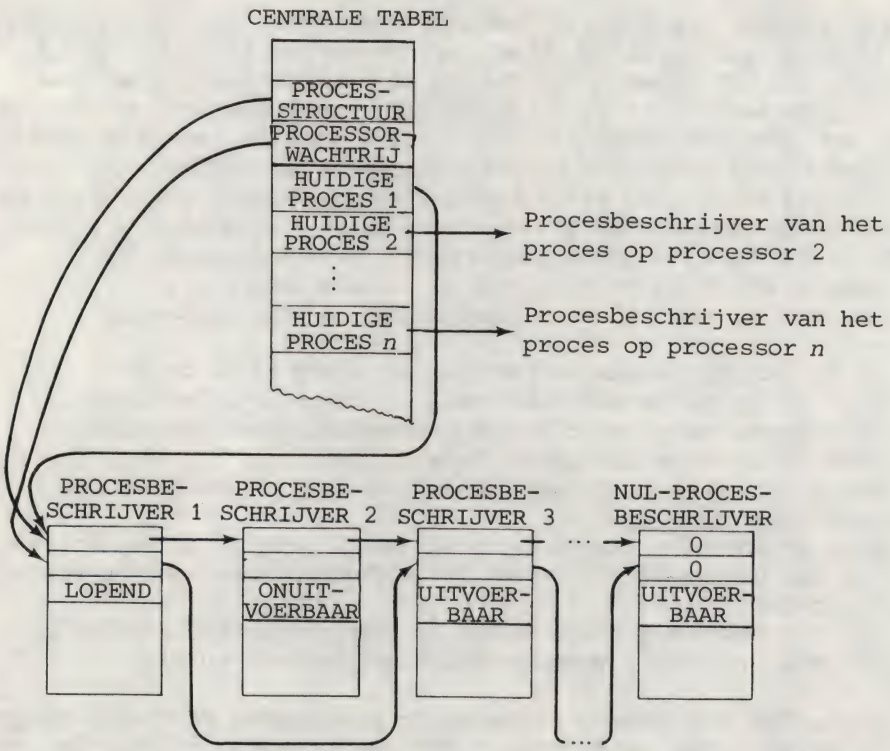
De werking van het verdeelprogramma is vrij eenvoudig:

(1) Is het huidige proces nog steeds het geschiktste om uit te voeren? Zo ja, laat dan de besturing verdergaan op het punt dat aangegeven wordt door de programmateller die opgeslagen is door de interrupthardware. Zo niet, dan .....
(2) Sla de vluchtige omgeving van het huidige proces op in zijn procesbeschrijver.
(3) Haal de vluchtige omgeving terug uit de programmabeschrijver van dat proces dat het meest in aanmerking komt om voortgezet te worden.
(4) Geef de besturing eraan terug, op die geheugenplaats die door de teruggehaalde programmateller aangegeven wordt.

Om te bepalen welk proces het meest in aanmerking komt om voortgezet te worden is het voldoende om aan alle uitvoerbare processen een prioriteitsvolgorde te geven. De toewijzing van de prioriteiten is geen functie van het verdeelprogramma, maar van de werkindeler, die in hoofdstuk 8 besproken wordt. Voor het moment merken we op dat de prioriteiten worden berekend aan de hand van factoren zoals het aantal benodigde hulpbronnen of de tijd die verstreken is sinds het proces de laatste keer liep of de relatieve importantie van de gebruiker die het proces gestart heeft. Voor wat het verdeelprogramma aangaat, zijn de prioriteiten *op voorhand* gegeven.

In ons papieren besturingssysteem voegen we alle procesbeschrijvers samen in een rij, die geordend is volgens afnemende prioriteit, zodat het meest voor de hand liggende proces vooraan staat. We noemen deze rij de *processorwachtrij*; hij wordt in figuur 4.4 geïllustreerd. Het is dus de taak van het verdeelprogramma om het eerste proces uit de processorwachtrij uit te voeren dat niet al op een andere processor loopt. Dit kan al dan niet hetzelfde proces zijn dat al liep voordat het verdeelprogramma aangeroepen werd.

We merken in het voorbijgaan op dat de introductie van een processorwachtrij betekent dat de actie, die door een interruptroutine ondernomen moet worden om een proces uitvoerbaar te maken, nu tweevoudig is. Ten eerste moet het de statusingang in de procesbeschrijver veranderen en ten tweede moet het de procesbeschrijver in de processorwachtrij invoegen op de plaats die aangegeven wordt door zijn prioriteit. In de volgende sectie zullen we zien dat deze



Figuur 4.4 *Processtructuur en processorwachtrij*

handeling gemakkelijk verricht kan worden door het uitvoeren van een *verhoog*handeling op een seinpaal waarop het betrokken proces een *passeer*handeling heeft uitgevoerd.

Het is natuurlijk mogelijk dat de processorwachtrij op een bepaald moment minder processen bevat dan er processoren zijn, bijvoorbeeld veroorzaakt doordat er meerdere processen zijn die op in- of uitvoer wachten. Deze situatie, die waarschijnlijk het gevolg is van een slechte indelingsbeslissing op hoog niveau, betekent dat er voor een aantal centrale processoren geen werk te doen is. Het heeft de voorkeur, boven het laten ronddraaien van een processor in het verdeelprogramma, een extra proces te introduceren dat de laagste prioriteit heeft en altijd uitvoerbaar is; dit heet het *nulproces*. Het nulproces kan slechts een lege lus zijn, of het kan nog een nuttige functie vervullen zoals het uitvoeren van testprogramma's voor de processor. Er zijn gevallen bekend waarin het nulproces gebruikt werd voor esoterische zaken, zoals het oplossen van eindspelen van schaakpartijen of voor het berekenen van de decimalen achter de komma van pi. Zijn positie achteraan de processorwachtrij is weergegeven in figuur 4.4.

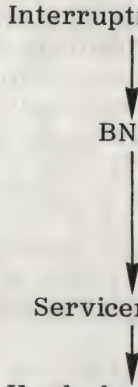
De werking van het verdeelprogramma kan nu als volgt samengevat worden:

(1) Is het huidige proces nog steeds het eerste niet-lopende proces in de processorwachtrij? Zo ja, ga er dan mee verder. Zo nee, dan .....
(2) Sla de vluchtige omgeving van het huidige proces op.
(3) Herstel de vluchtige omgeving van het eerste niet-lopende proces in de processorwachtrij.
(4) Hervat de uitvoering van dit proces.

Voordat we het verdeelprogramma verlaten moeten we nog opmerken dat we een erg eenvoudige structuur hebben gekozen voor het lopende proces. Veel besturingssystemen maken een onderverdeling in klassen van de uitvoerbare processen; onderscheid wordt gemaakt aan de hand van criteria als: benodigde hulpbronnen en faciliteiten, maximaal toegestane wachttijd, of de toegestane tijd voordat het verwijderd wordt. Zo heeft het DEC System-10 bijvoorbeeld drie processorwachtrijen, voor processen die respectievelijk 2,00, 0,25 en 0,2 seconden toegewezen krijgen voordat ze verwijderd worden. Iedere wachtrij wordt volgens het principe 'wie het eerst komt, die het eerst maalt' afgehandeld, terwijl de wachtrijen met de kortere verwerkingstijden een hogere prioriteit hebben. Een proces wordt in eerste instantie in de 0,02-seconde wachtrij geplaatst; als het zijn hele quantum toegestane tijd loopt, dan wordt het overgeplaatst naar de 0,25-seconde wachtrij, en overeenkomstig, als het gedurende zijn hele verhoogde quantum loopt, dan wordt het naar de 2-seconde wachtrij overgeplaatst. Het resultaat is dat ervoor gezorgd wordt dat processen, die te maken hebben met aan de computer gekoppelde terminals, waarvan een kenmerk is dat zij weinig eisen stellen aan de processor, snel bediend worden, terwijl

opdrachten die van de processor afhankelijk zijn langer maar minder frequent aandacht krijgen. We zullen hier meer over vertellen als we in hoofdstuk 8 scheduling (het maken van werkindelingen) bespreken.

We besluiten deze sectie met een samenvatting van de relatie tussen de BNIB en het verdeelprogramma in het hieronder staande diagram.



Interruptmechanisme	Sla de programmateller op Sla andere registers op (optie) Ga de BNIB in
BNIB	Sla de programmaregisters op (als dit hierboven nog niet gedaan is) Identificeer de interrupt (met behulp van hardware) Start een serviceroutine
Serviceroutine	Behandel de interrupt, mogelijk verandert de status van een proces
Verdeelprogramma	Is overschakelen van de processor nodig? Zo niet, hervat het onderbroken proces Sla de vluchtige omgeving op van het huidige proces Herstel de vluchtige omgeving van het eerste geschikte proces in de processorwachtrij Draag de besturing over aan het nieuwe proces

4.6 DE IMPLEMENTATIE VAN *passeer* EN *verhoog*

Het laatste deel van de kern is het implementeren van de een of andere vorm van een communicatiemechanisme tussen de processen. Zoals in hoofdstuk 3 is aangegeven zullen we de handelingen *passeer* en *verhoog* gebruiken als basiselementen voor onze communicatie, een keuze die gebaseerd is op het algemeen verspreide begrip en het gemak van implementatie van seinpalen. Andere communicatiemechanismen kunnen in de literatuur aangetroffen worden (Adrews en Schneider, 1983). Sommige hiervan, zoals monitoren (zie sectie 3.4), kunnen zelf geïmplementeerd worden op de manier van seinpalen.

Passeer en *verhoog* zijn in de kern opgenomen omdat:

(1) ze voor alle processen beschikbaar moeten zijn en daarom op een laag niveau geïmplementeerd moeten worden;

(2) de *passeer*handeling tot gevolg kan hebben dat een proces geblokkeerd wordt, wat het aanroepen van het verdeelprogramma veroorzaakt voor het opnieuw toewijzen van de processor. Daarom moet de *passeer*handeling toegang hebben tot het verdeelprogramma;
(3) een geschikte manier voor een interruptroutine voor het wakker maken (uitvoerbaar maken) van een proces is het uitvoeren van een *verhoog*handeling op een seinpaal, waarop het proces een *passeer*handeling heeft uitgevoerd. Daarom moet *verhoog* toegankelijk zijn voor de interruptroutines.

De handelingen die we moeten implementeren (zie hoofdstuk 3) zijn:

passeer(s) : **als** $s > 0$ **dan** verminder s

verhoog(s) : verhoog s

Hier is s een willekeurige seinpaal. We ontwikkelen onze implementatie als volgt.

(1) Blokkeren en deblokkeren

De *passeer*handeling houdt in dat processen geblokkeerd worden als een seinpaal de waarde 0 heeft, en bevrijd worden als een *verhoog*handeling de waarde verhoogt naar 1. De natuurlijke manier om dit te implementeren is het koppelen van een *seinpaal-wachtrij* aan elke seinpaal. Als een proces een 'niet-geslaagde' *passeer*handeling uitvoert (dat wil zeggen hij werkt op een seinpaal met de waarde 0), dan wordt het toegevoegd aan de seinpaalwachtrij en wordt het onuitvoerbaar gemaakt. Hier staat tegenover, dat als een *verhoog*handeling uitgevoerd wordt, er een proces uit de seinpaalwachtrij genomen kan worden (tenzij die leeg is) en weer uitvoerbaar gemaakt kan worden. De seinpaal moet daarom geïmplementeerd worden met twee componenten: een geheel positief getal en een wachtrijwijzer (die de waarde 0 mag hebben).

Op dit punt is onze implementatie als volgt:

passeer(s) : **als** $s \neq 0$ **dan** $s := s - 1$
zoniet dan voeg proces toe aan seinpaalwachtrij
en maak het onuitvoerbaar;

verhoog(s) : **als** wachtrij leeg is **dan** $s := s + 1$
zoniet dan haal een proces uit de seinpaalwachtrij en maak het uitvoerbaar;

Merk hierbij op dat de seinpaal niet binnenin *verhoog* verhoogd hoeft te worden, omdat het bevrijde proces onmiddellijk weer de seinpaal zou moeten verlagen bij het afmaken van zijn *passeer*handeling.

(2) Het plaatsen in en verwijderen uit wachtrijen

We hebben nog niets gezegd over welk proces het geluk heeft om uit de seinpaalwachtrij verwijderd te worden na een *verhoog*handeling. Ook hebben we nog niet gesteld of een proces dat aan de wachtrij wordt toegevoegd bij een niet-geslaagde *passeer*handeling, aan de kop, aan de staart of ergens in het midden toegevoegd moet worden. Met andere woorden, we hebben tot nu toe de organisatie van de wachtrij nog niet gespecificeerd.

Voor de meeste seinpalen voldoet het eenvoudige 'eerste in, eerste uit' (Engels: 'first in, first out', FIFO) omdat dit verzekert dat ten slotte alle processen bevrijd worden. Soms verdient het de voorkeur de wachtrij op een andere basis te ordenen, mogelijk aan de hand van een prioriteit vergelijkbaar met die, die in de processorwachtrij gebruikt is. Deze laatste vorm van organisatie zorgt ervoor dat processen met een hoge processorprioriteit geen lange perioden nutteloos blijven hangen in een seinpaalwachtrij. Het belangrijke punt is dat verschillende seinpalen een verschillende vorm van wachtrij-organisatie kunnen vereisen, zodat er een extra component betrokken moet worden in de implementatie van de seinpaal die aangeeft welke wachtrij-organisatie er van toepassing is. Deze component kan eenvoudig een gecodeerde beschrijving zijn van de wachtrij-organisatie of, in complexere gevallen, kan het een wijzer zijn naar een klein programmastukje dat de handelingen verzorgt voor het plaatsen in en het verwijderen uit de wachtrij. De structuur van een seinpaal in onze implementatie is weergegeven in figuur 4.5.

(3) Toewijzing van de processor

Zowel *passeer* als *verhoog* kunnen de status van een proces veranderen; de eerste door het ontuitvoerbaar te maken, de tweede door het omgekeerde. Er moet daarom een uitgang gemaakt worden naar het verdeelprogramma voor een beslissing over welk proces het volgende is dat uitgevoerd moet gaan worden. In gevallen waarin er geen processtatus veranderd is (dat wil zeggen als er een *passeer* op een seinpaal met een positieve waarde is uitgevoerd, of als er

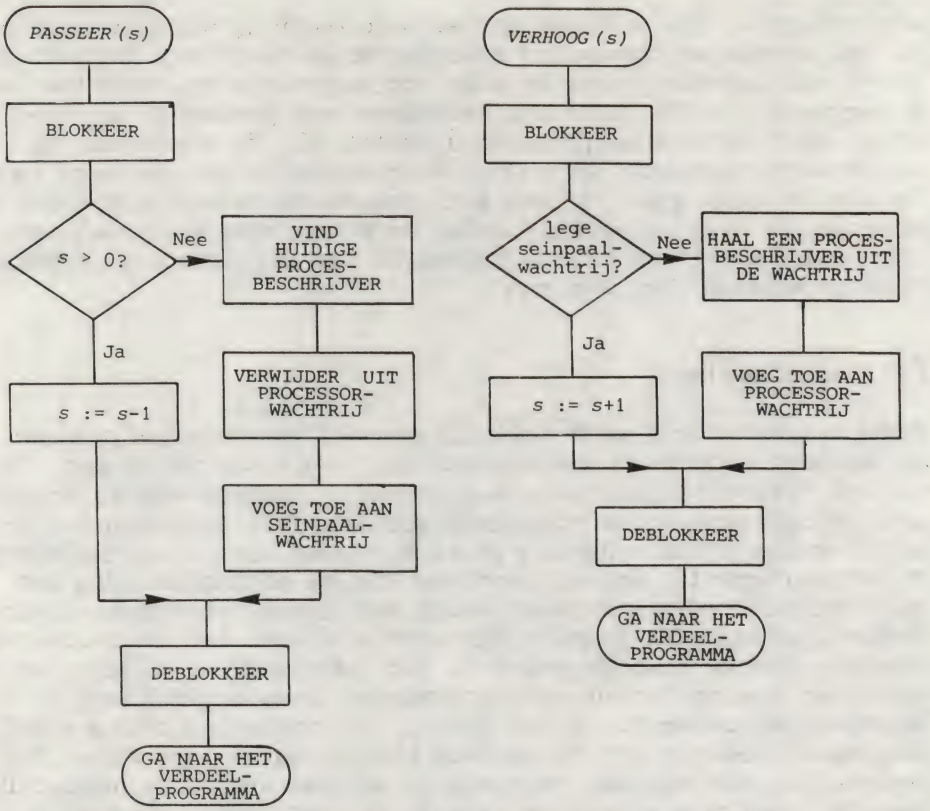
Figuur 4.5 *Structuur van een seinpaal*

een *verhoog* op een seinpaal met een lege wachtrij is uitgevoerd) zal het verdeelprogramma het huidige proces hervatten, daar dit het eerste niet-lopende proces is in de processorwachtrij. Er is wat voor te zeggen dat in dit geval het terugkeren van *passeer* of *verhoog* direct naar het huidige proces zou moeten zijn, in plaats van via het verdeelprogramma; de hieruit voortvloeiende toename in efficiëntie zou ten koste gaan van een iets complexere kode voor *passeer* en *verhoog*. In deze bespreking zullen we kiezen voor eenvoud boven efficiëntie, terwijl wij ons realiseren dat het omgekeerde standpunt even gerechtvaardigd kan zijn.

(4) Ondeelbaarheid

Zoals in hoofdstuk 3 werd duidelijk gemaakt moeten zowel *passeer* als *verhoog* ondeelbare handelingen zijn, in die zin dat op een moment slechts één proces ze mag uitvoeren. Daarom moeten ze alletwee geïmplementeerd worden als procedures, die beginnen met de een of andere *afsluit*handeling en eindigen met een *ontsluit*handeling. In een configuratie met een processor kan de afsluithandeling het eenvoudigst worden geïmplementeerd door blokkeren van het interruptmechanisme. Dit zorgt er met zekerheid voor dat een proces de controle over de centrale processor niet kan verliezen tijdens het uitvoeren van *passeer* of *verhoog* omdat er geen mogelijkheid is waardoor het onderbroken kan worden. De ontsluithandeling wordt uitgevoerd door het eenvoudig deblokkeren van de interrupts. Bij een machine met meerdere processoren is deze procedure ongeschikt, daar het voor twee processen mogelijk is tegelijkertijd *passeer* of *verhoog* in te gaan doordat zij op verschillende processoren lopen. In dit geval hebben we een ander machanisme nodig, zoals een 'test en zet' instructie. Dit is een instructie die de waarde van een geheugenplaats in een enkele handeling test en verandert. Gedurende de uitvoering van de instructie worden pogingen van andere processen de geheugenplaats te benaderen onderdrukt.

Het idee hierachter is dat een bepaalde geheugenplaats als vlag (een teken om aan te geven of een opdracht is uitgevoerd, vergelijkbaar met de variabele *poort* in sectie 3.3) gebruikt wordt, die aangeeft of toegang tot *passeer* en *verhoog* toegestaan is. Als de vlag (bijvoorbeeld) niet nul is, dan is de toegang geoorloofd, anders niet. De blokkeerhandeling bestaat uit een 'test en zet' instructie op de vlag, die de waarde ervan bepaalt en deze tegelijkertijd op nul zet. Als de waarde van de vlag niet nul is, dan gaat het proces verder, anders blijft het de 'test en zet' instructie uitvoeren totdat het proces, dat zich op dit moment in de *passeer*- of *vervolg*procedure bevindt, de vlag deblokkeert door hem op een waarde niet gelijk aan nul te zetten. Merk op dat moeilijkheden, die vergelijkbaar zijn met die welke optraden met betrekking tot de variabele *poort* in sectie 3.3, worden vermeden door het implementeren van de hele 'test en zet' instructie als een enkele ondeelbare handeling. Dit blokkeer/deblokkeermechanisme is toegepast op vele computers, zoals de IBM 370 serie.



Figuur 4.6 *De implementatie van passeer en verhoog*

Een alternatief voor de 'test en zet' instructie, toegepast op de Burroughs 6000 serie en de Plessey 250, is een instructie voor het onderling verwisselen van de inhoud van twee geheugenplaatsen. De blokkeerhandeling begint met het verwisselen van de waarde van een vlag en een geheugenplaats die van tevoren op nul gezet is. De waarde van deze tweede geheugenplaats wordt dan bekeken om te zien of toegang geoorloofd is, terwijl elk ander proces dat probeert toegang te krijgen de nul vindt die achtergebleven is door de verwisseling. Een proces dat na de verwisseling merkt dat de vlag nul is, herhaalt eenvoudig de blokkeerhandeling totdat de vlag ongelijk aan nul wordt gemaakt door een ander proces dat een deblokkeerhandeling uitvoert.

Opgemerkt moet worden dat deze twee mechanismen het gebruik inhouden van de een of andere vorm van *actief wachten*. Dat wil zeggen dat een proces, dat niet voorbij de blokkeerhandeling kan, zijn processor vasthoudt in een vaste lus waarin deze herhaaldelijk de handeling probeert totdat de vlag vrijkomt. Zolang de *passeer-* en *vervolg*procedures simpel zijn, zal de tijd die betrokken is bij het actieve wachten vrij kort zijn.

We benadrukken dat blokkeer- en deblokkeerhandelingen niet gebruikt kunnen worden als vervanging voor *passeer* en *verhoog*. Het actieve wachten of het interruptverbod, die beide gebruikt worden bij de implementatie van de blokkeerhandeling, zouden niet acceptabel zijn met betrekking tot de tijdsduur die processen kunnen worden vertraagd door een *passeer*handeling. De onderstaande tabel laat de inhoudelijk verschillende niveaus zien waarop de twee soorten handelingen bestaan.

	Passeer en *Verhoog*	Blokkeer en Deblokkeer
Doel	Algemene proces-synchronisatie	Wederzijdse uitsluiting van processen bij *passeer*- en *vervolg*procedures
Implementatie-niveau	Software	Hardware
Vertragings-mechanisme	Wachtrijen	Actief wachten/ interruptverbod
Typische vertragingstijd	Enkele seconden	Enkele microseconden

Het resultaat van de voorgaande discussie is dat we *passeer* en *verhoog* kunnen implementeren als twee gerelateerde procedures zoals we hebben laten zien in figuur 4.6. De procedures zullen worden aangeroepen door extracodes die een deel zijn van het instructierepertoire van alle processen.

We hebben nu de systeemkern voltooid, welke bestaat uit de Basis Niveau Ingreep Besturing, het verdeelprogramma en de procedures voor *passeer* en *verhoog*, die alle draaien in de supervisortoestand.

5 Het geheugenbeheer

Het geheugenbeheer is de volgende laag van onze 'ui'. We zetten het direct naast de kern omdat een proces weinig kan doen, tenzij het wat geheugen bezit. Het kan zeker niet uitgevoerd worden als er geen ruimte is voor het erbij behorende programma, het kan ook geen in- of uitvoer verzorgen zonder ruimte voor buffers.

In dit hoofdstuk zullen we als eerste de doelstellingen van geheugenbeheer bespreken en het begrip virtueel geheugen invoeren. We zullen manieren beschrijven waarop virtuele geheugens geïmplementeerd kunnen worden en we zullen dan diverse beleidslijnen voor de implementatie bespreken. Ten slotte zullen we zien hoe het geheugenbeheer in ons papieren besturingssysteem opgenomen kan worden.

5.1 DOELSTELLINGEN

De doelstellingen van geheugenbeheer zijn vijfvoudig.

(1) Het verplaatsen

In een meervoudig geprogrammeerde computer zal het beschikbare geheugen op elk gegeven moment gedeeld worden tussen een aantal processen en het is voor de individuele programmeur onmogelijk van tevoren te weten welke andere programma's in het geheugen aanwezig zullen zijn wanneer zijn eigen programma uitgevoerd wordt. Dit betekent dat, op het moment dat hij het programma schrijft, hij niet precies weet waar in het geheugen het programma gesitueerd zal zijn. Daardoor kan hij zijn programma niet in termen van absolute geheugenplaatsen schrijven. Als het aan zijn programma toegewezen geheugen gedurende de gehele uitvoering hetzelfde zou blijven, dan zou het natuurlijk mogelijk zijn om, op het moment dat het programma geladen wordt, de symbolische of relatieve adressen (= manier om geheugenplaats aan te geven) om te zetten in absolute adressen, maar in de praktijk is dit zelden het geval. Als processen naar hun einde lopen, komt de ruimte die ze gebruiken vrij voor andere

processen en het kan nodig zijn met processen in het geheugen te schuiven, waarbij het beste gebruik van de geheugenruimte voorop staat. Het kan, in het bijzonder, wenselijk zijn zo met processen te schuiven dat kleine ongebruikte gebieden vrij geheugen samengetrokken kunnen worden tot een enkel groter gebied dat beter bruikbaar is. Zodoende kan het aan een proces toegewezen geheugendeel tijdens zijn levensduur veranderen en het systeem moet ervoor zorgen dat de adressen die door de programmeur gebruikt werden omgezet worden naar de feitelijk adressen waarin het proces fysiek gesitueerd is.

(2) Bescherming

Als meer processen het geheugen delen is het natuurlijk essentieel voor hun adequate werking dat aan geen ervan toegestaan is de inhoud van geheugenplaatsen, die op dat moment aan een ander proces zijn toegewezen, te veranderen. Controle van de adressen tijdens het vertalen van het programma geeft niet voldoende bescherming, omdat de meeste talen een dynamische berekening toestaan van de adressen tijdens het uitvoeren, bijvoorbeeld door het berekenen van reeksvermeldingen of van (ver-) wijzers naar gegevensstructuren. Daarom moeten alle verwijzingen die door een proces gegenereerd worden, gecontroleerd worden tijdens de uitvoering, om te verzekeren dat zij alleen refereren aan geheugenruimte die aan dat proces is toegewezen. (Strikt genomen hoeft alleen toegang waarbij geschreven wordt gecontroleerd te worden, maar als privacy van informatie gevraagd wordt, dan moet ook toegang waarbij gelezen wordt gecontroleerd worden.)

(3) Gemeenschappelijk gebruik

Ondanks de behoefte aan bescherming zijn er gelegenheden waarbij diverse processen toegang moeten kunnen krijgen tot hetzelfde geheugendeel. Bijvoorbeeld, als een aantal processen hetzelfde programma uitvoert: het heeft dan voordelen elk proces toegang te geven tot dezelfde kopie van het programma, boven het hebben van een eigen kopie van elk programma. Hiermee is het geval vergelijkbaar, waarin processen samen gebruik maken van een gegevensstructuur en zodoende gemeenschappelijk toegang hebben tot het geheugengebied dat dit bevat. Het beheersysteem van het geheugen moet daarom gecontroleerde toegang tot gemeenschappelijke geheugendelen toestaan, zonder dat afbreuk gedaan wordt aan essentiële bescherming.

(4) Logische organisatie

De traditionele computer heeft een één-dimensionale of lineaire geheugenruimte en de adressen zijn van nul af opeenvolgend genummerd tot de bovengrens van het geheugen. Terwijl het wel een natuurgetrouwe afspiegeling is van de hardware van de machine, geeft dit de manier waarop programma's gewoonlijk geschreven zijn niet echt goed weer. De meeste programma's zijn op de een of andere manier gestructureerd - in modulen of procedures - en verwijzen naar afzonderlijke gegevensgebieden, die al dan niet te wijzigen zijn. Bijvoorbeeld, een vertaalprogramma kan geschreven zijn met afzonderlijke modulen voor lexicografische analyse, syntaxis analyse en code aanmaak, en bij zijn gegevensgebieden kan een tabel zitten met gereserveerde woorden (niet te wijzigen) en een symbolentabel (die gewijzigd wordt als er nieuwe identificaties tegengekomen worden tijdens het vertalen). Als de logische afdelingen in programma en gegevens afgebeeld zijn op een overeenkomstige *segmentatie* van de geheugenruimte, dan vloeit daar een aantal voordelen uit voort. Ten eerste is het mogelijk segmenten onafhankelijk te coderen zodat alle verwijzingen van het ene segment naar het andere door het systeem tijdens de uitvoering ingevuld kunnen worden; ten tweede is het, met weinig extra moeite, mogelijk om aan verschillende segmenten een verschillende mate van bescherming te geven (bijvoorbeeld: alleen lezen, alleen uitvoeren); en ten derde is het mogelijk mechanismen in te voeren waarmee segmenten gemeenschappelijk gebruikt kunnen worden door verschillende processen.

(5) Fysieke organisatie

Historisch gezien hebben de algemene behoefte aan grote hoeveelheden opslagruimte en de hogere kosten van snelle geheugens geleid tot een bijna universele invoering van opslagsystemen met twee niveaus. Het compromis tussen snelheid en kosten wordt juist bereikt door het combineren van een kleine hoeveelheid (tot een paar millioen bytes) direct toegangkelijk hoofdgeheugen met een veel grotere hoeveelheid (mogelijk enkele honderden millioenen bytes) secundair geheugen of achtergrondopslag. Het hoofdgeheugen maakt gebruik van halfgeleidertechnologie en heeft een benaderingstijd in de orde van grootte van 100 nanoseconden; het achtergrondgeheugen is meestal gebaseerd op magnetische schijven, met een benaderingstijd tot 100 milliseconden.

Omdat in de meeste gewone systemen alleen de informatie in het hoofdgeheugen direct benaderd kan worden, is de informatiestroom tussen het primaire en secundaire geheugen duidelijk van het hoogste belang. Het is natuurlijk mogelijk om dit de individuele programmeur te laten organiseren, zodat deze de verantwoordelijkheid heeft voor het, naar behoefte, heen en weer schuiven van programma- of gegevenssecties, van of naar het hoofdgeheugen. In de eerste dagen van de computer was dit inderdaad het geval, en de techniek van

het *overlappende programmeren* - het schrijven van programmadelen die in het geheugen elkaar overlappen - werd een kunst op zich. Er zijn echter twee goede redenen die tegen een dergelijke benadering pleiten. De eerste is dat de programmeur niet veel moeite wil besteden aan het schrijven van overlappingen: hij of zij is te geïnteresseerd in het oplossen van zijn eigen problemen om tijd te verspillen aan bijkomende vraagstukken. Intelligente vertaalprogramma's kunnen hulp bieden door het automatisch aanmaken van overlapcodes op de juiste plaatsen in het programma, maar in de meeste gevallen heeft het vertaalprogramma niet voldoende informatie om te weten wanneer de overlappingen nodig zijn. De tweede reden is dat de programmeur, ten gevolge van het dynamische hergebruik van het geheugen, ten tijde van het schrijven niet weet hoeveel ruimte er in het geheugen beschikbaar is en waar die ruimte zich bevindt. Daardoor wordt het overlappend programmeren onuitvoerbaar.

Uit deze redenen blijkt duidelijk dat het schuiven van informatie tussen de twee geheugenniveaus een verantwoordelijkheid moet zijn van het systeem. Hoe deze verantwoordelijkheid ten uitvoer kan worden gebracht wordt verderop in dit hoofdstuk besproken.

5.2 HET VIRTUELE GEHEUGEN

De doelstellingen die hierboven zijn opgesomd kunnen bereikt worden door het inhoudelijk simpele (en elegante) gebruik van een vertalingsmechanisme voor de adressen of *adresvertaler* voor het omzetten van de adressen, die door de programmeur gebruikt zijn, in de bijbehorende fysieke geheugenplaatsen die feitelijk aan het programma toegewezen zijn. Het onderhouden van de adresvertaling is een systeemfunctie en mogelijke implementatievormen worden in de volgende sectie besproken. Op dit moment is het cruciale punt het onderscheid tussen *programma-adressen* - adressen die door de programmeur zijn gebruikt - en de *fysieke geheugenplaatsen* waarin zij geplaatst worden. De reeks programma-adressen staat bekend als *adresruimte* (of *name space*); de reeks geheugenplaatsen in de computer is de *geheugenruimte*. Als de adresruimte aangegeven wordt door N en de geheugenruimte door M, dan kan de adresvertaling aangegeven worden met

$$f : N \rightarrow M$$

In de meest tegenwoordig voorkomende computers is de geheugenruimte lineair - dat wil zeggen dat de geheugenplaatsen opeenvolgend genummerd zijn vanaf nul - en de grootte ervan gelijk is aan de hoeveelheid hoofdgeheugen die in de configuratie aanwezig is. We zullen zien dat de adresruimte echter niet lineair hoeft te zijn en dat het, afhankelijk van de betreffende implementatie van de adresvertaler, kleiner dan, gelijk aan, of groter dan de geheugenruimte kan zijn.

Een andere manier om de adresvertaler te bekijken is deze te beschouwen als een middel dat de programmeur in staat stelt gebruik te maken van een reeks programma-adressen die totaal verschillend kunnen zijn van de beschikbare geheugenplaatsen. Zodoende 'ziet' en programmeert de programmeur een *virtueel geheugen*, waarvan de karakteristieken verschillen van die van het werkelijke geheugen. De adresvertaler is zo ontworpen dat het een virtueel geheugen produceert dat geschikt is voor de programmeur en op die manier bereiken we enkele of alle doelen die in de vorige sectie opgesomd werden.

5.3 DE IMPLEMENTATIE VAN HET VIRTUELE GEHEUGEN

(1) Basis- en grensregisters

De eerste twee doelstellingen die in sectie 5.1 zijn opgesomd, namelijk verplaatsing en bescherming, kunnen als volgt bereikt worden met betrekkelijk eenvoudige adresvertalers.

Als een proces in het geheugen wordt geladen, wordt de laagst gebruikte geheugenplaats in een *basisregister* (of *datumregister*) geplaatst en alle programma-adressen worden als relatief aan dit *basisadres* beschouwd. De adresvertaling bestaat dus eenvoudig uit het optellen van de programma-adressen bij het basisadres om tot de corresponderende geheugenplaats te komen; dat wil zeggen:

$$f(a) = B + a$$

Hierin is a het programma-adres en B is het basisadres van het proces.

Verplaatsing wordt eenvoudig bereikt door het verplaatsen van het proces en het herwaarderen van het basisadres naar de geschikte waarde. De bescherming van geheugenruimte tussen processen onderling kan bereikt worden door het inschakelen van een tweede register, het *grensregister*, dat het adres bevat van de hoogste geheugenplaats die een bepaald proces mag benaderen. De adresvertaling (zie figuur 5.1) die uitgevoerd wordt door de adresseerhardware gaat dan verder volgens het onderstaande schema:

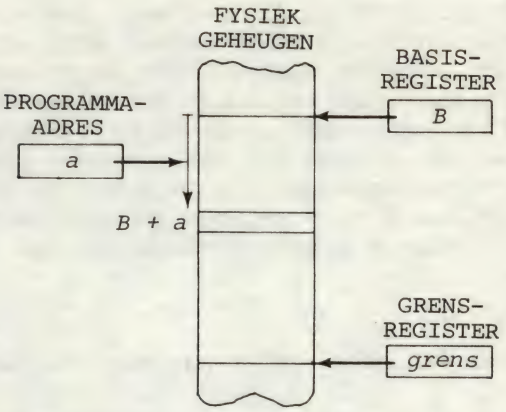
(a) **als** $a < 0$ **dan** geheugenschending

(b) $a' := B + a$

(c) **als** $a' >$ *grens* **dan** geheugenschending

(d) a' is de gevraagde geheugenplaats

Opgemerkt moet worden dat de adresruimte die op deze manier vertaald wordt lineair is en dat de grootte, die het verschil tussen de



Figuur 5.1 *Adresvertaling voor basis- en grensregisters*

basis- en de grensregisters, noodzakelijkerwijs kleiner dan of gelijk is aan de geheugenruimte. Een gevolg hiervan is dat de doelstellingen (3) en (4) uit sectie 5.1 niet bereikt worden.

Om ervoor te zorgen dat het vertalen van de adressen niet te veel tijd gebruikt, moeten de basis- en grensregisters in snelle hardware geïmplementeerd zijn. De kosten van de registers kunnen verminderd worden en de vertaling kan versneld worden door het verwijderen van de bits met een lage orde, wat inhoudt dat de grootte van de adresruimte een veelvoud van 2^n moet zijn (hier is n het aantal verwijderde bits).

Een kleine variatie op het basis-grens schema is het gebruik van het grensregister voor het bevatten van de *geheugenlengte* in plaats van zijn bovengrens. In dit geval is de adresvertaling:

(a) **als** $a < 0$ **of** $a >$ *lengte* **dan** geheugenschending

(b) $a' := B + a$

(c) a' is de gewenste geheugenplaats

Het voordeel is dat de grenscontrole niet afhankelijk is van het resultaat van de optelling, dus kunnen de controles en de optelling parallel geschieden. (De optelling wordt afgebroken als er een geheugenschending plaatsvindt.)

Om economische redenen is het onuitvoerbaar een stel basis- en grensregisters aan elk proces, dat in het geheugen aanwezig kan zijn, te geven. In plaats daarvan wordt er voorzien in een enkel stel registers voor iedere processor en die worden geladen met de basis- en grensadressen van het huidig lopende proces. Deze waarden vormen een deel van de vluchtige omgeving van het proces en worden opgeslagen als het proces onderbroken wordt.

Het delen van programma's die door meer processen kunnen worden binnengekomen, kan bereikt worden door het voorzien in twee paar registers in plaats van één. Eén paar wordt gebruikt om de geheugenruimte aan te geven, die in beslag genomen wordt door de opnieuw binnenkomende code en de waarden daarin zijn hetzelfde voor alle processen die de code gebruiken; het andere paar wordt gebruikt om de gegevensgebieden aan te geven die bij de code horen en bevatten verschillende waarden voor elk erbij betrokken proces. Het oudere DEC System-10 is een voorbeeld van een machine die deze techniek gebruikt.

(2) Het indelen van het geheugen in pagina's (de paginaverdeling)

In het basis-grens schema, dat hierboven beschreven is, is de grootte van de adresruimte noodzakelijkerwijs kleiner dan of gelijk aan de geheugenruimte. Als we een programmeur een virtueel geheugen willen geven dat groter is dan het beschikbare fysieke geheugen, waarmee we hem of haar verlossen van de last van het schrijven van overlappingen, dan moeten we een adresvertaling bedenken die schijnbaar het onderscheid tussen hoofdgeheugen en secundair geheugen doet verdwijnen. Het idee van de *opslag op één niveau*, waarin het secundaire geheugen als een verlenging van het hoofdgeheugen wordt voorgesteld, werd voor het eerst op de Atlas computer op de universiteit van Manchester geïntroduceerd rond 1960 en heeft sindsdien een verregaande invloed gehad op het ontwerpen van computers.

De één-niveau opslag kan gerealiseerd worden door het *indelen van het geheugen in pagina's (blokken)*, waarbij de virtuele adresruimte in pagina's van gelijke grootte ingedeeld wordt (bij de Atlas 512 woorden), en het hoofdgeheugen wordt, hiermee vergelijkbaar, in *paginakaders* van dezelfde grootte ingedeeld. De paginakaders worden verdeeld onder de processen die op een moment in het systeem aanwezig zijn, zodat op ieder moment een gegeven proces een paar pagina's in het hoofdgeheugen aanwezig heeft (zijn *actieve* pagina's) en de rest in het secundaire geheugen aanwezig is (zijn *inactieve* pagina's). Het mechanisme van de paginaverdeling heeft twee functies:

(a) Het uitvoeren van de adresvertaling; dat wil zeggen het bepalen aan welke pagina een programma-adres refereert en het vinden van het paginakader dat de pagina (eventueel) bezet.
(b) Het overbrengen van pagina's uit het secundaire geheugen naar het hoofdgeheugen als dat verlangd wordt en het weer terugbrengen naar het secundaire geheugen als zij niet langer meer gebruikt worden.

We beschrijven deze functies opeenvolgend.

Om vast te stellen naar welke pagina een programma-adres verwijst, worden de hogere-orde bits geïnterpreteerd als het

paginanummer en worden de lagere-orde bits geïnterpreteerd als het woordnummer in de pagina. Als dus de paginagrootte 2^n is, dan vertegenwoordigen de voorste n bits van het adres het woordnummer en de resterende bits het paginanummer. Het totale aantal bits in het adres is voldoende voor de adressering van het gehele virtuele geheugen. Op de Atlas, bijvoorbeeld, was het programma-adres 20 bits lang; dit gaf een virtueel geheugen van 2^{20} woorden; de paginagrootte was 512 woorden (2^9), en dus gaven de voorste 9 bits het woordnummer aan en de laatste 11 bits het paginanummer. Het totale aantal pagina's in het virtuele geheugen was daarom 2^{11} (dit in tegenstelling tot de 32 pagina's in het oorspronkelijke fysieke geheugen).

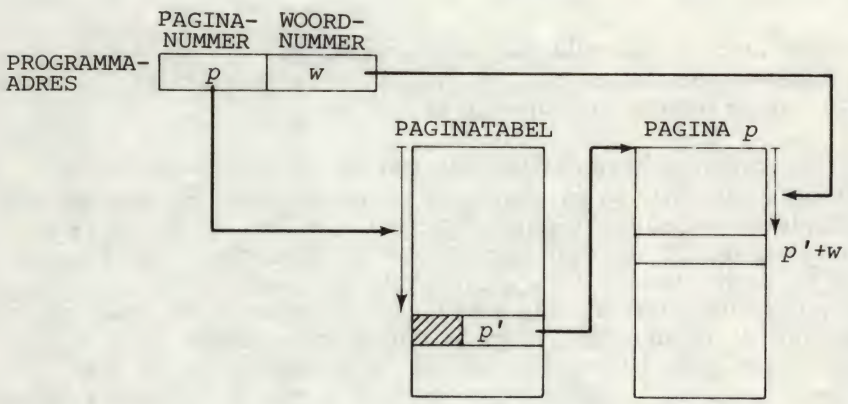
We willen graag benadrukken dat de onderverdeling van adressen in woorden en pagina's een hardwarefunctie is en dat het voor de programmeur vanzelfsprekend is; wat hem betreft is hij aan het programmeren in één grote aaneengesloten adresruimte.

De adresvertaling van het pagina- en woordnummer naar het fysieke geheugen wordt door middel van een *paginatabel* gemaakt, de p-de ingang ervan bevat geheugenplaats p' van het paginakader dat paginanummer p bevat. De mogelijkheid dat de p-de pagina niet in het hoofdgeheugen aanwezig is wordt zo direct besproken. Het woordnummer, w, wordt aan p' toegevoegd om de gevraagde geheugenplaats te krijgen (zie figuur 5.2).

De adresfunctie is daarom:

$$f(a) = f(p,w) = p' + w$$

Hierin zijn het programma-adres a, het paginanummer p en het woordnummer w gerelateerd aan de paginagrootte Z door:



Figuur 5.2 *Eenvoudige adresvertaler voor het verdelen in pagina's*

p = integraal deel van (a/Z)

w = restwaarde van (a/Z)

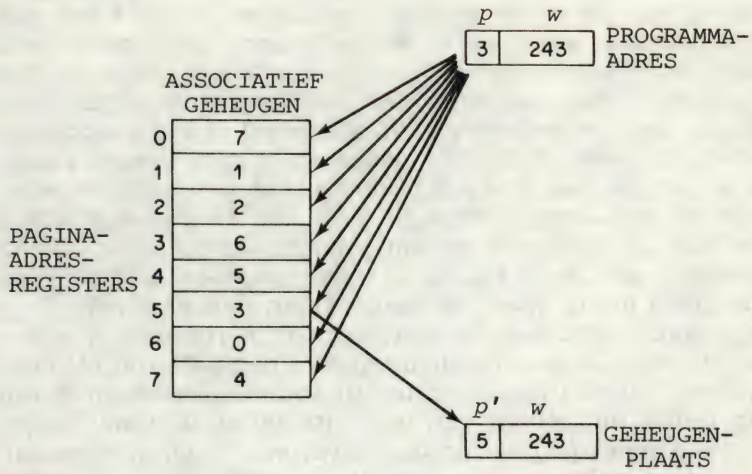
Omdat het aantal paginakaders (hoeveelheid werkelijk geheugen) dat aan een proces is toegewezen gewoonlijk minder zal zijn dan het aantal pagina's dat het feitelijk gebruikt, is het heel wel mogelijk dat het programma-adres verwijst naar een pagina die op het moment niet in het hoofdgeheugen gehouden wordt. In dit geval zal de bijbehorende ingang in de paginatabel leeg zijn en wordt er een 'paginafout' interrupt gegeven, als er geprobeerd wordt deze te benaderen. Deze ingreep zorgt ervoor dat het paginaverdelingsmechanisme de overdracht start van de ontbrekende pagina van het secundaire geheugen naar het hoofdgeheugen en zorgt voor het overeenkomstig bijwerken van de paginatabel. Het huidige proces wordt onuitvoerbaar gemaakt totdat de overdracht voltooid is. De plaats van de pagina in het secundaire geheugen kan in een aparte tabel of in de paginatabel zelf opgeslagen worden. In het tweede geval is er een 'aanwezigheidsbit' nodig bij elke ingang van de paginatabel, om aan te geven of de pagina in het hoofdgeheugen aanwezig is of niet en of het adresveld beschouwd moet worden als een paginakader of als een plaats in het achtergrondgeheugen.

Als er geen leeg paginakader bestaat op het moment dat de paginafout voorkomt, moet er een andere pagina naar het secundaire geheugen verplaatst worden om ruimte vrij te maken voor de binnenkomende pagina. De keuze welke pagina op deze manier uitgewisseld moet worden komt voort uit een *pagina-omzet-algoritme* (Engels: *page turning algorithm*); we bespreken de diverse algoritmen in volgende secties. Voor het ogenblik merken we op dat de informatie die nodig is voor het pagina-omzet-algoritme, opgeslagen kan worden in een paar bits die toegevoegd zijn aan elke paginatabel-ingang (het gearceerde gebied van figuur 5.2). Deze informatie zou kunnen zijn:

(a) hoeveel keer er naar de pagina verwezen is;
(b) de laatste keer dat er naar de pagina verwezen werd;
(c) of er op de pagina geschreven is.

Er moet misschien opgemerkt worden dat de hele adresvertaling door de hardware verzorgd wordt, behalve wanneer een pagina uit het secundaire geheugen ingebracht moet worden. In dat geval wordt de toepassing van het pagina-omzet-algoritme en het bijwerken van de paginatabel verzorgd door de software.

De voorgaande bespreking geeft een algemene schets van de werking van de paginaverdeling; in de praktijk moeten er diverse veranderingen gemaakt worden. In het bijzonder wordt in het beschreven systeem de tijd, die benodigd is voor iedere verwijzing naar het geheugen, in feite verdubbeld door de noodzakelijkheid eerst de paginatabel te benaderen. Een manier om dit te omzeilen zou het opslaan van de paginatabel in een stel snelle registers zijn,



Figuur 5.3 *Vertaling met behulp van een associatief geheugen*

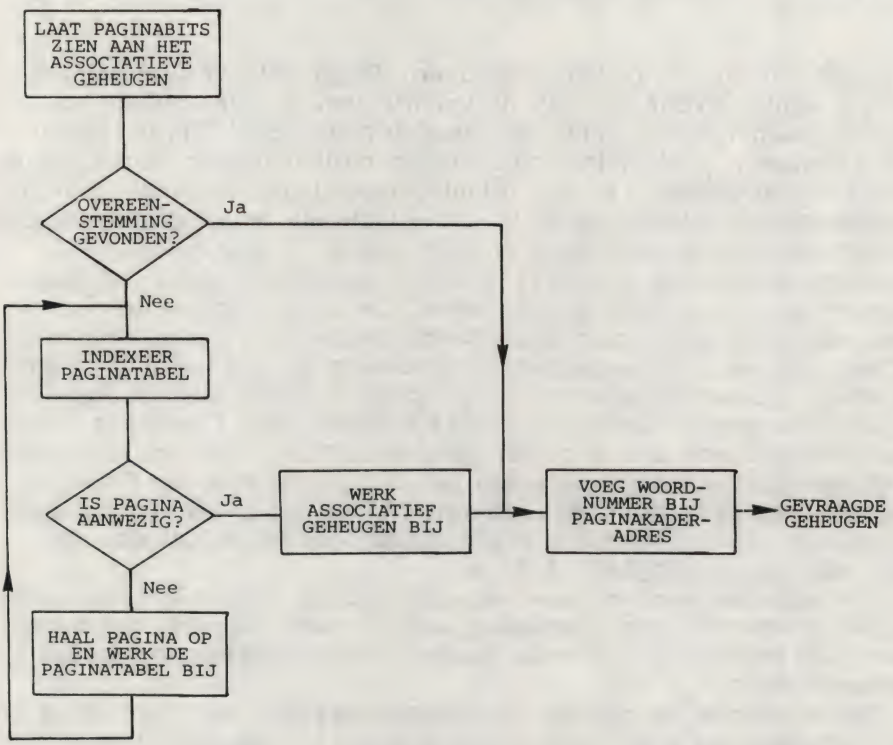
in plaats van in het gewone geheugen. De grootte van de paginatabel is echter evenredig aan de grootte van de adresruimte en daarom zou het aantal benodigde registers te groot zijn om economisch haalbaar te zijn. De oplossing voor dit probleem wordt gevonden in het toepassen van een totaal verschillende techniek voor het benaderen van actieve pagina's. Deze techniek houdt de toevoeging van een *associatief geheugen* in, dit bestaat uit een kleine set *pagina-adresregisters* (PARs), waarvan elk afzonderlijk het paginanummer van een actieve pagina bevat. De PARs hebben de eigenschap dat ze *tegelijkertijd* onderzocht kunnen worden op de aanwezigheid van een paginanummer dat voorkomt in een bepaald programma-adres. Zo wordt bijvoorbeeld programma-adres 3243 in figuur 5.3 gesplitst in paginanummer 3 en woordnummer 243. (Voor het gemak wordt aangenomen dat de paginagrootte 1000 is.) Het paginanummer wordt dan gelijktijdig vergeleken met de inhoud van alle PARs en blijkt te corresponderen met die van PAR 5. Dit geeft aan dat paginanummer 3 op het ogenblik paginakader 5 bezet en dat dus de gevraagde geheugenplaats 5243 is.

Het gebruik van een associatief geheugen vermindert de extra ruimte die voor de adresvertaling nodig is met ongeveer net zoveel ruimte als benodigd is voor de opslag van een paginatabel in het hoofdgeheugen.

Om ervoor te zorgen dat alle actieve pagina's een verwijzing van een PAR hebben, heeft men evenveel PARs als paginakaders in het geheugen nodig. In systemen met kleine geheugens (zoals de Atlas) is dit mogelijk, maar bij grotere systemen is het niet economisch rendabel om in alle benodigde PARs te voorzien. (Er is echter te verwachten dat de economische argumenten zullen veranderen met de

ontwikkeling van de techniek.) In dergelijke gevallen kan men een compromis bereiken door het in het geheugen houden van een hele paginatabel voor elk proces en door het gebruik van een klein associatief geheugen voor het verwijzen naar een paar pagina's van de meest recent actieve processen. In dit geval is het paginakader, waarnaar door iedere PAR verwezen wordt, niet langer vanzelfsprekend in de positie van de PAR binnenin het associatieve geheugen, maar moet als een extra veld in de PAR zelf toegevoegd worden. De hardware die zorgt voor de geheugenadressering voert dan de adresvertaling uit die in figuur 5.4 weergegeven is. Software interventie is alleen nodig voor het verplaatsen van pagina's.

Een probleem dat niet is weergegeven in figuur 5.4 is het onderscheiden in het associatief geheugen van pagina's die bij het huidige proces horen, en van pagina's die bij andere processen horen. Een oplossing is het uitbreiden van de PARs zodat zij zowel de procesidentificatie als het paginanummer bevatten. Ieder proces dat aan het associatieve geheugen aangeboden wordt moet dan zowel procesidentificatie als paginabits bevatten. Een alternatief is het uitbreiden van de PARs met een enkel bit, dat op één gezet wordt bij ingangen



Figuur 5.4 *Adresvertaling voor paginaverdeling met een klein associatief geheugen*

die bij de huidige processen horen, en dat elders op nul gezet wordt. Deze oplossing houdt noodzakelijkerwijs in dat er voor iedere processor in de configuratie een apart associatief geheugen is. (Dit is waarschijnlijk sowieso aan te raden om ervoor te zorgen dat de logica van de geheugenadressering niet de beperkende factor wordt tussen processoren en geheugen.)

Vanzelfsprekend is het wenselijk dat het associatieve geheugen de paginanummers bevat van die pagina's waarvan het het meest waarschijnlijk is dat ernaar verwezen wordt. Helaas is er geen algemeen algoritme om hier met zekerheid voor te zorgen (vergelijk sectie 5.4 betreffende het verkeer van pagina's tussen hoofgeheugen en secundaire geheugens); in de praktijk is het associatieve geheugen gewoonlijk cyclisch gevuld met de adressen van die pagina's waarnaar het meest recent verwezen is. Dit vrij grove algoritme blijkt behoorlijk effectief; afhankelijk van de geheugengrootte kan er verwacht worden dat een opslag van slechts 8 of 16 registers een gemiddeld scoringspercentage heeft dat groter is dan 99%.

Als laatste opmerking over het mechanisme van 'het verdelen in pagina's' willen we wijzen op de grote hoeveelheid hoofdgeheugen die door paginatabellen zelf in beslag genomen kan worden. Op de versie van het DEC System-10 met paginaverdeling bijvoorbeeld bestaat de adresruimte uit 512 pagina's van elk 512 woorden; dit houdt een paginatabel van 256 woorden in voor elk proces dat in het geheugen aanwezig is (twee ingangen worden gebundeld tot een woord). In werkelijkheid is de paginatabel in een van zijn eigen pagina's geplaatst; hiervan wordt de andere helft gebruikt voor het opslaan van de procesbeschrijver.

De grootte van de paginatabellen kan teruggebracht worden tot het aantal feitelijk gebruikte pagina's, dit in plaats van het aantal dat in de adresruimte aanwezig is, door ze in stukjes te benaderen in plaats van via een index. Het idee hierachter is dat de paginatabel niet een ingang voor iedere pagina in de adresruimte bevat, maar slechts verwijst naar die pagina's die gebruikt zijn. Elke keer dat er een nieuwe pagina in het hoofdgeheugen wordt gebracht, wordt het paginanummer en het bijbehorende paginakaderadres in de tabel ingevoerd op een punt dat bepaald wordt door een passende verdeelfunctie. Dezelfde verdeelfunctie kan dan gebruikt worden om de plaats van de pagina's te bepalen gedurende de adresvertaling. De prijs die betaald wordt voor het in grootte terugbrengen van de paginatabel is een toename in de geheugenbenaderingstijd bij sommige verwijzingen. Die toename is het gevolg van twee factoren: ten eerste is er tijd nodig voor het berekenen van de verdeelfunctie zelf en ten tweede kan het nodig zijn, in het geval dat de verdeelde ingangen in de paginatabel met elkaar conflicteren, de tabel meer dan eens te benaderen voordat de plaats van de gevraagde pagina is gevonden. Het schema kan desondanks goed werkbaar gemaakt worden door het gebruik van een associatief geheugen dat het aantal gevallen waarin naar de paginatabel verwezen wordt tot ongeveer 1% van het totaal kan terugbrengen en door de hardware de verdeling te laten uitvoeren.

(3) Segmentatie

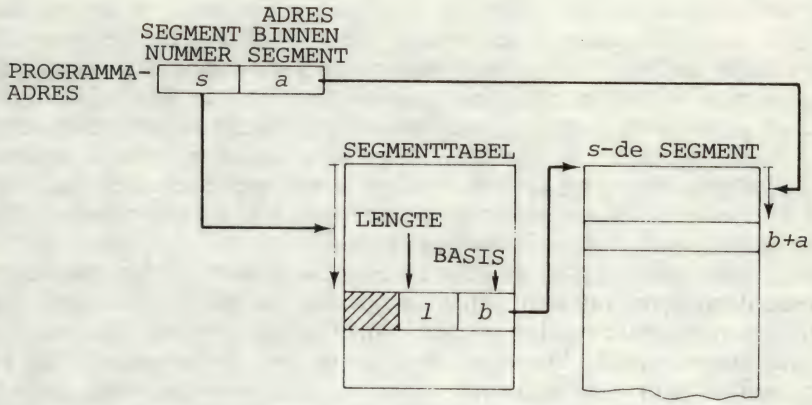
De derde doelstelling van sectie 5.1, te weten het zodanig organiseren van de adresruimte dat het de logische verdelingen in programma's en gegevens weergeeft, kan door het gebruik van de *segmentatietechniek* bereikt worden. Het idee hierachter is dat de adresruimte in *segmenten* onderverdeeld wordt, waarbij ieder segment overeenkomt met een procedure, programmamodule of gegevensverzameling. Dit kan op een eenvoudige manier verwezenlijkt worden door het toevoegen van een aantal paren basis- en grensregisters aan iedere processor, zodat de adresruimte in een aantal gebieden onderscheiden kan worden. De Modular One computer heeft bijvoorbeeld drie paar registers en de drie segmenten worden achtereenvolgens gebruikt voor: het programma, gegevens behorende bij een proces en gegevens die door meerdere processen gebruikt kunnen worden. De nadelen van dit eenvoudige mechanisme zijn dat het aantal segmenten, om economische redenen, klein is en dat er vooraf afgesproken moet zijn welke segmenten er voor welk doel gebruikt worden.

Een meer flexibele opstelling van het virtuele geheugen is dat de programmeur toegestaan wordt een groot aantal segmenten naar believen te gebruiken en hem/haar de mogelijkheid te geven naar segmenten te verwijzen door middel van namen die hij/zij er zelf aan gegeven heeft. Hierdoor wordt de adresruimte twee-dimensionaal omdat de afzonderlijke programma-adressen geïdentificeerd worden door het geven van zowel een segmentnaam als een adres in het segment. (In werkelijkheid vervangt het besturingssysteem, om de implementatie te vergemakkelijken, de segmentnaam door een uniek segmentnummer als er de eerste keer naar het segment verwezen wordt.) Een algemeen programma-adres bestaat daarom uit een paar (s,a) waarin s het segmentnummer en a het adres in het segment is.

De adresvertaling kan geïmplementeerd worden door middel van een *segmenttabel* voor ieder proces, waarvan de s-de ingang de basisplaats en de lengte van het s-de segment van het proces bevat. De ingangen in de segmenttabel worden soms omschreven als *segmentbeschrijvers (segment descriptors)*. In vereenvoudigde vorm gaat het vertalen als volgt (zie figuur 5.5):

(a) Zoek programma-adres (s,a) op
(b) Gebruik s voor indexering van segmenttabel
(c) Als $a < 0$ of $a > l$ dan geheugenschending
(d) $(b + a)$ is de gevraagde geheugenplaats

De bescherming tegen geheugenschendingen wordt verzorgd door het vergelijken van het woordadres met de segmentlengte l. Een bijkomende bescherming kan gegeven worden door het inbedden van een aantal *beschermingsbits* in elke segmentbeschrijver (gearceerd weergegeven in figuur 5.5), die de manieren aangeven waarop het bijbehorende segment kan worden benaderd.



Figuur 5.5 *Eenvoudige adresvertaling voor segmentatie*

Merk op dat een segment probleemloos door een aantal processen gemeenschappelijk gebruikt kan worden. Het enige dat hiervoor nodig is, is een beschrijver voor het betrokken segment in de segmenttabel van ieder proces. De beschermingsbits in iedere beschrijver kunnen verschillen, zodat het ene proces leestoegang heeft tot een gedeeld segment terwijl een ander proces in staat is erin te schrijven. Op die manier maakt de segmentatie een flexibel gemeenschappelijk gebruik mogelijk van programma's en gegevens voor processen.

We benadrukken hier dat, ondanks de schijnbare overeenkomst van figuren 5.2 en 5.5, de adresvertaling voor paginaverdeling en segmentatie in de volgende opzichten totaal verschillend zijn:

(a) Het doel van de segmentatie is de logische verdeling van de adresruimte; het doel van de paginaverdeling is de fysieke verdeling van het geheugen voor de implementatie van een 'één-niveau opslag'.
(b) Pagina's hebben een vastgestelde grootte die door de machine-architectuur bepaald is; segmenten kunnen elke willekeurige grootte hebben die door de gebruiker bepaald wordt (tot een maximum dat bepaald wordt door de manier waarop programma-adressen onderverdeeld worden in segment- en woordnummers).
(c) De verdeling van programma-adressen in pagina- en woordnummers is een hardwarefunctie en het te hoog worden van het woordnummer heeft automatisch de verhoging van het paginanummer tot gevolg; de verdeling in segment- en woordnummers is een logische, en er bestaat geen overloop van woordnummers naar segmentnummers. (Als het segmentnummer te hoog wordt, wordt er een geheugenschending gegenereerd.)

In de praktijk is het vertalen van het adres niet zo eenvoudig als de hierbovenstaande beschrijving doet denken. De belangrijkste complicatie is dat het bij grote programma's niet mogelijk kan blijken alle segmenten in het hoofdgeheugen te houden, in het bijzonder omdat het geheugen door meer processen gedeeld zal worden. Feitelijk is dit een situatie waarin het virtuele geheugen groter is dan het fysieke geheugen en er kan mee gewerkt worden door het toepassen van een paginaverdeling of door het naar behoefte uitwisselen van hele segmenten uit het geheugen.

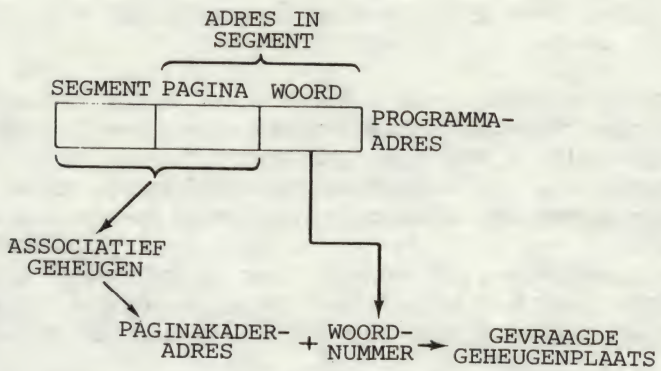
Als er een indeling in pagina's toegepast wordt, dan bestaat elk segment meestal uit een aantal pagina's en heeft zijn eigen paginatabel. Op voorwaarde dat er een aantal pagina's van het segment in het geheugen staat, verwijst de ingang van de segmenttabel naar de paginatabel van het segment; zo niet, dan is deze leeg. De adresvertaling die door de adresseerhardware wordt uitgevoerd, is als volgt:

(a) Zoek programma-adres (s,a)
(b) Gebruik s voor de indexering van de segmenttabel
(c) **Als** de s-de ingang leeg is, **dan** maak een nieuwe (lege) paginatabel, **zoniet** haal dan het adres uit de paginatabel
(d) Verdeel het woordadres a in paginanummer p en woordnummer w
(e) Gebruik p voor de indexering van de paginatabel
(f) **Als** de p-de ingang leeg is, **dan** haal een pagina uit de achtergrondopslag, **zoniet** zoek dan het adres p' op uit het paginakader
(g) Voeg p' toe aan woordnummer w om de gevraagde geheugenplaats te krijgen

De stappen (a) tot en met (c) geven de vertaling gedurende de segmentatie weer; stappen (d) tot en met (g) (die overeenkomen met figuur 5.2) geven de vertaling weer ten gevolge van het indelen in pagina's van de segmenten.

De extra geheugenverwijzingen die nodig zijn voor het benaderen van de segment- en paginatabellen, kunnen vermeden worden door het gebruik van een associatief geheugen op een manier die vergelijkbaar is met die, welke beschreven is voor het alleen maar indelen in pagina's. In dit geval bevat elke ingang in het associatief geheugen zowel de segment- als de paginanummers van de laatst benaderde pagina's. De segment- en paginabits van ieder programmaadres worden aan het associatief geheugen aangeboden (figuur 5.6) en als er overeenstemming gevonden wordt, dan wordt het woordnummer toegevoegd aan het bijbehorende paginakaderadres om zo de gevraagde geheugenplaats te verkrijgen. Alleen wanneer er geen overeenstemming gevonden wordt, wordt de gehele handeling zoals hierboven beschreven erop toegepast.

Als het indelen in pagina's niet wordt toegepast, dan zijn de ingangen in de segmenttabellen zoals in figuur 5.5 weergegeven. De foutmelding bij het ontbreken van een segment zorgt ervoor dat het hele gevraagde segment in het hoofdgeheugen gebracht wordt; het



Figuur 5.6 *Het vertalen van adressen bij in pagina's verdeelde segmenten met gebruik van een associatief geheugen*

kan zijn dat een ander segment verbannen wordt naar de achtergrondopslag om voor voldoende ruimte te zorgen. Het beleid dat toegepast wordt bij het toewijzen van geheugen aan segmenten zal in de volgende sectie beschreven worden.

De voordelen van het gebruik van pagina's voor de implementering van een gesegmenteerde adresruimte zijn tweeledig. Ten eerste hoeft niet het gehele segment in het geheugen aanwezig te zijn - slechts die pagina's die op het ogenblik gebruikt worden moeten beschikbaar zijn; ten tweede is het niet nodig dat de pagina's aaneenvolgende gebieden in het geheugen beslaan - ze kunnen over het gehele geheugen verdeeld worden al naar gelang er geschikte paginaruimte gevonden wordt. Tegenover het indelen in pagina's moet de ingewikkeldheid van de adresvertaling en de extra last van de paginatabellen gezet worden. Voorbeelden van systemen die in pagina's ingedeelde segmenten hebben zijn de Honeywell 645, de ICL 2900 en de grote IBM 370 machines; die met segmenten, die niet in pagina's onderverdeeld zijn, zijn de Burroughs machines (bijvoorbeeld de B6700) en de PDP-11/45.

5.4 HET BELEID BIJ HET TOEWIJZEN VAN GEHEUGENRUIMTE

De besprekingen in de vorige sectie hadden alleen betrekking op het *mechanisme* van het geheugenbeheer en niet op het *beleid* voor de besturing ervan. In deze sectie zullen wij ons concentreren op het beleid dat het mogelijk maakt de mechanismen goed toe te passen.

Het beleid voor goed geheugenbeheer is grofweg in drie categorieën in te delen:

(1) *Vervangingsbeleid*, dat bepaalt welke informatie in het hoofdgeheugen verwijderd kan worden en zo voor niet toegewezen (vrije) delen in het geheugen zorgt.
(2) *Ophaalbeleid*, dat bepaalt wanneer informatie in het hoofdgeheugen geladen moet worden, bijvoorbeeld op verzoek of op voorhand.
(3) *Plaatsingsbeleid*, dat bepaalt waar de informatie in het hoofdgeheugen geplaatst moet worden, dat wil zeggen dat dit een deelverzameling kiest van een willekeurig niet toegewezen gebied.

We zullen zien dat het beleid van type (2) bij in pagina's ingedeelde systemen vrijwel hetzelfde is als bij niet in pagina's ingedeelde systemen. Het beleid van de vormen (1) en (3) verschilt echter bij systemen met of zonder pagina-indeling: de verschillen vloeien voort uit de tegenstelling van pagina's met een vaste grootte tegenover de variabele grootte van de informatieblokken waarmee gewerkt moet worden bij systemen die geen pagina-indeling hebben.

(1) Het plaatsingsbeleid bij systemen zonder pagina-indeling

In deze subsectie bekijken we de gegevensoverdracht naar het hoofdgeheugen bij gevallen waar de virtuele geheugenruimte òf gesegmenteerd is òf geïmplementeerd is met behulp van basisgrensparen. In beide gevallen zullen we de informatieblokken die overgebracht worden met het woord 'segment' aanduiden, met de wetenschap dat bij het gebruik van een enkelvoudig basisgrenssysteem, een segment dan de gehele adresruimte van een proces zal zijn.

De situatie is dat er van tijd tot tijd segmenten in het hoofdgeheugen ingevoegd of eruit verwijderd worden. (Het invoegen of verwijderen vindt respectievelijk plaats op aangeven van het ophaal- en vervangingsbeleid.) Als het systeem in evenwicht is (dat wil zeggen dat gedurende een tijd de ruimte die door het verwijderen vrij komt gelijk is aan die welke opgevuld wordt door het invoegen) lijkt het geheugen op het dambord uit figuur 5.7.

Het is duidelijk dat het plaatsingsbeleid een lijst moet bijhouden van de plaats en de grootte van de 'gaten' die niet toegewezen zijn; we zullen dit de *gatenlijst* noemen. Zijn taak is het beslissen welk gat er gebruikt moet worden en het bijwerken van de gatenlijst na

Figuur 5.7 *Gesegmenteerd geheugen*

iedere invoeging. Als het te plaatsen segment kleiner is dan het te gebruiken gat, dan wordt het segment aan de 'linker- of onderkant' van het gat geplaatst. Deze tactiek minimaliseert de mate waarin het gat wordt versnipperd. Als het segment echter groter is dan welk gat dan ook, heeft het plaatsingsbeleid als bijkomende taak de segmenten, die al in het geheugen aanwezig zijn, zo te verplaatsen dat er een gat ontstaat dat groot genoeg is voor zijn doel.

Er zijn in de literatuur talrijke plaatsingsalgoritmen beschreven, bijvoorbeeld Knuth (1968); we zullen hier vier principieel verschillende geven. In elk van de gevallen wordt de grootte van de gaten aangeduid met $x_1, x_2, \ldots, x_n$.

(a) Beste passing (best fit)

De gaten staan opgesomd in volgorde van toenemende grootte, dat wil zeggen dat $x_1 \leqq x_2 \leqq \ldots \leqq x_n$. Als s de grootte is van het te plaatsen segment, zoek dan de kleinste i waarvoor geldt dat $s \leqq x_i$.

(b) Slechtste passing (worst fit)

De gaten staan opgesomd in volgorde van afnemende grootte, dat wil zeggen dat $x_1 \geqq x_2 \geqq \ldots \geqq x_n$. Plaats het segment in het eerste gat en zet de overblijvende ruimte op de juiste plaats in de lijst terug.

(c) Eerste passing (first fit)

De gaten staan opgesomd in volgorde van toenemend basisadres. Vind de kleinste i waarvoor geldt dat $s \leqq x_i$. Het gebruik van het algoritme leidt na verloop van tijd tot een opeenhoping van kleine gaten aan de kop van de lijst. Om onnodig lange zoektijden bij grote segmenten te voorkomen, kan de beginpositie na iedere zoekactie één element in de cyclus doorgeschoven worden.

(d) Maatjes (Buddy)

Voor dit algoritme moet de grootte van de segmenten een macht van 2 zijn, dat wil zeggen dat $s = 2^i$ voor elke i kleiner dan een maximum k. Er wordt een aparte lijst bijgehouden voor iedere grootte $2^1, 2^2, \ldots, 2^k$. Een gat kan uit de $(i+1)$-lijst gehaald worden door het in tweeën te splitsen, waarbij een paar 'maatjes' gemaakt wordt op de i-lijst, met de grootte 2^i. Het algoritme voor het vinden van een gat met de grootte 2^i is recursief.

```
procedure haal gat (i);
begin als i = k dan fout;
      als i-lijst leeg is dan
      begin haal gat (i + 1)
            splits gat in maatjes;
            zet maatjes op i-lijst
      einde;
      neem het eerste gat in de i-lijst
einde
```

Het kan de argwanende lezer vreemd lijken dat algoritme (a) en (b) beide goed bruikbaar zijn, omdat zij gebruik maken van volstrekt tegengestelde strategieën betreffende het ordenen van de gatenlijst. Algoritme (a) spreekt mogelijk intuïtief meer aan, omdat het de verspilling in elk gat dat het selecteert lijkt te minimaliseren (dat wil zeggen dat deze het kleinst mogelijke gat selecteert dat voldoet). Algoritme (b) werkt echter met de filosofie dat er bij de toewijzing van een groot gat een gat overblijft dat groot genoeg is om in de toekomst nog nuttig te zijn, terwijl er bij de toewijzing van een klein gat een nóg kleiner gat resteert dat waarschijnlijk totaal nutteloos zal zijn.

Een probleem dat voorkomt bij alle vormen van plaatsingsbeleid is de *versnippering* - het opsplitsen van vrije geheugengaten in gaten die zo klein zijn dat er geen nieuwe segmenten ingepast kunnen worden. Als dit voorkomt moet de een of andere vorm van *verdichting* van het geheugen plaatsvinden, dat wil zeggen dat alle segmenten naar beneden verplaatst moeten worden, zodat er een groot gat aan de bovenkant van het geheugen verschijnt. De verdichting is het gemakkelijkst te bereiken als de gaten in adresvolgorde in een lijst zijn opgeslagen, zoals in het eerste algoritme. Een alternatieve benadering is het verdichten van het geheugen na iedere keer dat er een segment verwijderd is, zodoende wordt de versnippering helemaal voorkomen; de extra last, die het veelvuldig verdichten met zich meebrengt, wordt goedgemaakt door het feit dat men geen hele lijst af hoeft te zoeken als er een segment geplaatst moet worden.

(2) Plaatsingsbeleid bij systemen met paginaverdeling

Het plaatsingsbeleid bij systemen met een paginaverdeling is veel eenvoudiger dan bij systemen die geen paginaverdeling hebben; voor het plaatsen van k pagina's moeten we simpelweg een vervangingsbeleid hebben dat k paginakaders vrij maakt. De versnippering, in de betekenis die beschreven is voor systemen zonder paginaverdeling, kan niet optreden daar alle pagina's en alle paginakaders dezelfde grootte hebben. Een systeem met paginaverdeling kan echter te leiden hebben van een andere vorm van versnippering, *interne versnippering*, die optreedt omdat de ruimte die een proces nodig heeft gewoonlijk niet een exact veelvoud is van de paginagrootte. Als gevolg daarvan wordt een deel van het laatst toegewezen paginakader meestal verspild: het is te verwachten dat de gemiddelde verspilling een halve pagina groot is. Als de adresruimte bestaat uit diverse, in pagina's verdeelde, segmenten, wordt de mate van interne versnippering vermenigvuldigd met het aantal segmenten die het hoofdgeheugen op dat moment bevat.

(3) Het vervangingsbeleid voor systemen met paginaverdeling

Het is de taak van het vervangingsbeleid te beslissen welke informatieblokken naar het secundaire geheugen verwezen moeten worden, als er ruimte in het hoofdgeheugen gevonden moet worden voor nieuwe blokken. In deze subsectie bekijken we het geval waarin de informatieblokken pagina's zijn; we zullen later zien dat hetzelfde beleid, met de juiste aanpassingen, toegepast kan worden in de situatie zonder paginaverdeling.

Het ideaal is dat men die pagina vervangt, waarnaar de langste tijd in de toekomst niet verwezen gaat worden. Helaas is kennis over de toekomst moeilijk te krijgen: het beste wat men kan doen is het gedrag, zoals dat waarschijnlijk in de toekomst zal zijn, afleiden uit het gedrag zoals dat in het verleden was. De nauwkeurigheid van de gevolgtrekkingen hangt van de voorspelbaarheid van het programmagedrag af, we zullen dat nader bespreken in de sectie over ophaalbeleid.

Drie veel gebruikte vervangingsalgoritmen zijn:

(a) Minst Onlangs Gebruikt (MOG)

'Vervang de pagina die recent het minst is gebruikt.' De aanname is dat het toekomstige gedrag nauwkeurig het afgelopen gedrag zal volgen. De extra last is dat de volgorde van benadering van alle pagina's opgeslagen moet worden.

(b) Minst Frequent Gebruikt (MFG)

'Vervang de pagina die, gedurende een bepaalde tijd die hieraan net voorafgaat, het minst frequent gebruikt is.' De rechtvaardiging is dezelfde als bij (a) en de extra last is dat er voor iedere pagina een 'gebruiksgetal' bijgehouden moet worden. Een schaduwzijde is, dat een onlangs geladen pagina in het algemeen een laag gebruiksgetal zal hebben en ten onrechte vervangen zal worden. Een manier om dit te omzeilen is het blokkeren van de vervanging van pagina's die in het laatste tijdsinterval geladen zijn.

(c) Eerste-in, eerste-uit (First in, first out)

'Vervang de pagina die het langst aanwezig is geweest.' Dit is een eenvoudiger algoritme, waarvan de extra last alleen het opslaan van de volgorde van het laden van de pagina's is. Het houdt geen rekening met de mogelijkheid dat er naar de oudste pagina het intensiefst verwezen kan worden.

Simulatiestudies (Colin, 1971) hebben een prestatieverschil (uitgedrukt in het aantal overbrengingen dat nodig is voor het uitvoeren van een aantal opdrachten) aangetoond, dat varieert met het soort opdrachten dat uitgevoerd wordt, maar de variatie is zelden meer

dan 15%. Algoritme (c) presteert in het algemeen slechter dan de andere twee.

Ten slotte is het het opmerken waard dat pagina's waarop niet geschreven is niet in het secundaire geheugen teruggezet moeten worden, als we ervan uitgaan dat daar al een kopie bestaat. Of er op een bepaalde pagina geschreven is kan in een enkele bit, in de bijbehorende ingang in de paginatabel, bijgehouden worden.

(4) Vervangingsbeleid bij systemen zonder paginaverdeling

In deze subsectie bekijken we het beleid voor de situatie waarin de informatieblokken die verplaatst moeten worden segmenten in de ruimste zin van het woord zijn, zoals hiervoor in (1) besproken.

Het hoofddoel is hetzelfde als bij systemen met paginaverdeling, dat wil zeggen het vervangen van het segment waarnaar in de naaste toekomst het minst waarschijnlijk verwezen gaat worden. Men kan daarom verwachten dat hetzelfde beleid toepasbaar is en dat is ook het geval, maar met één belangrijke wijziging. De wijziging vloeit voort uit het feit dat niet alle segmenten dezelfde hoeveelheid geheugen innemen, daarom wordt de overweging welk segment er naar het secundaire geheugen verbannen moet worden, beïnvloed door de grootte van het te plaatsen segment. Als er een klein segment in het hoofdgeheugen gebracht moet worden, hoeft er maar een klein segment verplaatst te worden; hier staat tegenover dat de plaatsing van een groot segment de verplaatsing van een ander groot segment (of meerdere kleintjes) inhoudt.

Het is waarschijnlijk het eenvoudigste algoritme om dat enkele segment te verplaatsen dat, samen met de eraan grenzende gaten, genoeg ruimte vrijmaakt voor het binnenkomende segment. Als er meerdere van die segmenten zijn, dan kan een van de beleidsvormen die eerder besproken werden, zoals MOG, gebruikt worden om te beslissen welke er verplaatst moet worden. Als er geen enkel segment groot genoeg is om voldoende ruimte te maken, moeten meerdere segmenten verplaatst worden: een mogelijke keuze is de kleinste groep aaneengrenzende segmenten, die de benodigde ruimte vrijmaakt.

Het gevaar dat in zo'n algoritme schuilt is, dat er korte tijd later weer naar het segment dat (of de segmenten die) voornamelijk vanwege de grootte vervangen is (zijn), verwezen kan worden. Het gevaar kan verminderd worden door het segment zuiver op een (bijvoorbeeld) MOG-basis te selecteren, maar omdat de gekozen segmenten waarschijnlijk niet aan elkaar grenzen zal de een of andere vorm van verdichting nodig zijn. De lezer zal gemakkelijk kunnen begrijpen dat de relatieve waarde, die gehecht wordt aan de segmentgrootte, aan de verwachte verwijzingen in de toekomst en aan de verdichting, kan leiden tot een scala van complexe algoritmen, die moeilijk in te schatten zijn, behalve in het gebruik. We zullen hier niet verder op het onderwerp ingaan, maar verwijzen de belangstellende lezer naar gedetailleerdere studies, zoals die van Knuth (1968) en van Denning (1970).

(5) Ophaalbeleid

De vormen van ophaalbeleid bepalen wanneer er een informatieblok van het secundaire geheugen naar het hoofdgeheugen overgebracht wordt. De redenen voor het kiezen van een beleidsvorm zijn grofweg hetzelfde ongeacht of de blokken nu pagina's of segmenten zijn (in de ruime betekenis die we gebruikt hebben). De vormen van ophaalbeleid kunnen in twee grote klassen ingedeeld worden: *op verzoek* en *anticiperend*. De vormen van ophaalbeleid die 'op verzoek' leveren, halen de blokken op als ze nodig zijn; de vormen van 'anticiperend' ophaalbeleid halen ze op voorhand op.

Vormen van het 'op verzoek' beleid zijn duidelijk eenvoudiger te implementeren - de foutmelding dat een blok (pagina of segment) niet aanwezig is genereert een verzoek om deze op te gaan halen en het plaatsings- en/of vervangingsbeleid wijst geheugen toe aan het nieuwe blok. In systemen zonder paginaverdeling (bijvoorbeeld de Burroughs machines) is het overbrengen van blokken meestal op verzoek, hoewel dit ook in enkele systemen met paginaverdeling (bijvoorbeeld de Atlas) het geval is.

Anticiperende beleidsvormen vertrouwen op voorspellingen over het toekomstige programmagedrag om goed te kunnen werken. De voorspelling kan op twee zaken gebaseerd zijn:

(a) de manier waarop de programma's geconstrueerd zijn;
(b) gevolgtrekkingen uit het gedrag van het proces in het verleden.

Laten we (a) eens bekijken.

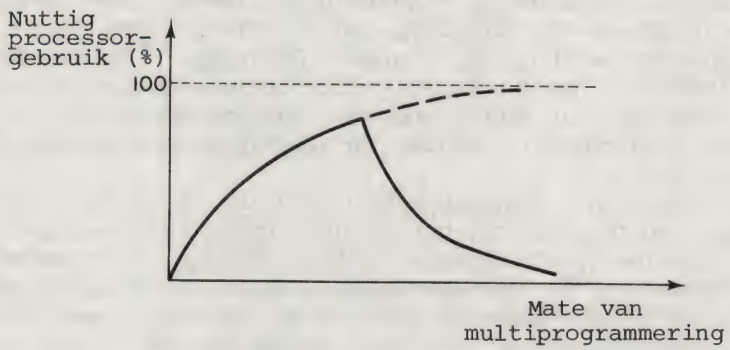
Veel programma's vertonen een gedrag dat bekend staat als *werkend in context*; dat wil zeggen dat een programma, in een willekeurig klein tijdsinterval, de neiging heeft binnen een bepaalde logische module te werken, terwijl het zijn instructies uit een procedure en zijn gegevens uit een gegevensgebied haalt. Zodoende hebben de programmaverwijzingen de neiging gegroepeerd te worden in kleine plaatselijke stukjes adresruimte. De plaatselijkheid wordt versterkt door het regelmatig voorkomen van lussen; hoe kleiner de lus, des te kleiner de spreiding van de verwijzingen. Bestudering van dit gedrag heeft tot de aanname (Denning, 1970) van het zogenaamde *plaatselijkheidsprincipe* geleid: 'programmaverwijzingen hebben de neiging in kleine plaatsen adresruimte gegroepeerd te worden en deze plaatsen hebben de neiging slechts met tussenpozen te veranderen'.

In hoeverre het plaatselijkheidsprincipe opgaat verschilt van programma tot programma; het zal bijvoorbeeld beter opgaan voor programma's die opeenvolgende reeksen (arrays) aanspreken dan voor programma's die complexe gegevensstructuren aanspreken. Het principe wordt door Denning gebruikt in samenhang met een geheugen met paginaverdeling, voor het formuleren van het *werkset* model van het programmagedrag, dat we in de volgende sectie kort zullen beschrijven.

5.5 HET WERKSET MODEL

Het werkset model van het programmagedrag (Denning, 1968) is een poging een kader te scheppen om een beter begrip van de prestaties van systemen met een paginaverdeling te krijgen; dit speelt in een situatie waarin met meervoudige programmering wordt gewerkt. De beleidsvormen betreffende het geheugenbeheer, die we in de vorige sectie hebben besproken, zijn gebaseerd op een bestudering van het procesgedrag in een geïsoleerde omgeving; ze houden geen rekening met eventuele gevolgen die kunnen voortkomen uit het tegelijkertijd aanwezig zijn van meerdere processen in de machine. Het 'vechten' om geheugenruimte blijkt te kunnen leiden tot gedrag dat niet zou optreden wanneer ieder proces apart zou lopen.

Als voorbeeld van wat er kan gebeuren bekijken we een systeem met één processor en een in pagina's verdeeld geheugen, waarbij de mate van meervoudige programmering (multiprogrammering), dat wil zeggen het aantal processen dat in het geheugen aanwezig is, geleidelijk opgevoerd wordt. Als de mate van meervoudige programmering toeneemt, zou men kunnen verwachten dat het nuttig gebruik van de processor ook toeneemt; dit omdat het verdeelprogramma altijd een grotere kans heeft dat het een programma vindt dat uitgevoerd kan worden. Waarneming bevestigt dit in het algemeen, zolang echter de mate van meervoudige programmering onder een bepaald niveau gehouden wordt, afhankelijk van de beschikbare geheugengrootte. Als echter de mate van de meervoudige programmering boven dit niveau uit gaat, dan ziet men (zie figuur 5.8) dat er een opvallende toename is in het paginaverkeer tussen het hoofdgeheugen en het secundaire geheugen, wat gepaard gaat met een vrij plotselinge afname van het nuttige processorgebruik. Hiervoor is de verklaring: de hoge mate van meervoudige programmering maakt het onmogelijk om voor elk proces voldoende pagina's in het geheugen te houden, teneinde het genereren van een groot aantal



Figuur 5.8 *Verschrotting*

paginafouten te voorkomen. Dit betekent dat het kanaal naar de achtergrondopslag zo verzadigd kan worden dat de meeste processen geblokkeerd wachten op een paginatransport, en dat de processor onderbezet wordt. Naar deze stand van zaken wordt vaak verwezen met de passende term *verschrotten* (Engels: *trashing*).

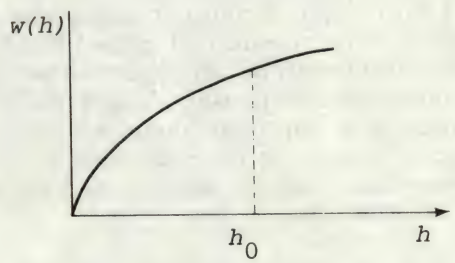
De les die uit deze illustratie geleerd kan worden is, dat ieder proces er behoefte aan heeft dat een bepaald minimum aantal pagina's, dat zijn *werkset* genoemd wordt, in het hoofdgeheugen gehouden wordt, voordat het effectief gebruik kan maken van de centrale processor. Als er minder dan dit aantal aanwezig zijn, dan zal het proces voortdurend onderbroken worden door paginafouten, die bijdragen aan de verschrotting. Om verschrotting te voorkomen is het nodig dat de mate van meervoudige programmering niet groter is dan dat niveau waarop de werksets van alle processen in het hoofdgeheugen gehouden kunnen worden.

De vraag doet zich nu voor, hoe je moet vaststellen welke pagina's deel uitmaken van de werkset van een proces. Het antwoord hierop is, dat de recente geschiedenis van het proces bekeken moet worden en dat er een beroep gedaan moet worden op de plaatselijkheid, zoals in de vorige sectie besproken werd. Formeler kan men stellen dat de werkset van een proces op tijdstip t gedefinieerd kan worden door:

$$w(t,h) = \{\text{pagina } i \mid \text{pagina } i \in N \text{ en pagina } i \text{ verschijnt in de laatste } h \text{ verwijzingen}\}$$

Met andere woorden: de werkset is die set pagina's waarnaar recent verwezen is, waarbij de 'recentheid' een van de parameters (h) van de set is. Aan de hand van het plaatselijkheidsprincipe zou men verwachten dat het lidmaatschap van de werkset langzaam met de tijd verwisselt. Denning heeft aangetoond dat de te verwachten grootte $w(h)$ van de werkset varieert met h, zoals in figuur 5.9 aangegeven is.

Als h vergroot wordt (dat wil zeggen hoe verder men in het verleden kijkt), verwacht men ook minder extra pagina's in de werkset. Dit maakt het mogelijk dat men een redelijk kleine waarde



Figuur 5.9 *De verwachte grootte van de werkset*

voor h (bijvoorbeeld h_0) vaststelt, met de wetenschap dat een grotere waarde van h de werkset niet wezenlijk zou vergroten.

Voor wat het ophaalbeleid en het vervangingsbeleid aangaat, ligt de waarde van de werkset in de volgende regel besloten:

> 'Voer een proces alleen dan uit als zijn gehele werkset in het hoofdgeheugen aanwezig is, en verban nooit een pagina die deel uitmaakt van de werkset van het een of andere proces.'

Alhoewel de werkset van een proces vrij arbitrair gedefinieerd kan zijn en alhoewel het plaatselijkheidsprincipe bij sommige programma's niet opgaat, kan het toepassen van de bovenstaande regel een aanzienlijke bijdrage leveren aan het voorkomen van verschrotting. Het model van de werkset wordt algemeen gebruikt, hoewel de definitie ervan wat verschilt van besturingssysteem tot besturingssysteem.

Het is belangrijk zich te realiseren dat de hierboven gegeven regel meer is dan alleen maar een beleid voor het geheugenbeheer, omdat deze een correlatie inhoudt tussen geheugen- en processortoewijzing. Tot nu toe hebben we in dit hoofdstuk het geheugenbeheer behandeld als een onderwerp dat losstaat van het beheer van andere hulpbronnen en faciliteiten. In werkelijkheid kan een hulpbron niet altijd bekeken worden zonder dat er rekening gehouden wordt met andere hulpbronnen, maar we zullen tot hoofdstuk 8 afzien van een volledige behandeling van de toewijzing van hulpbronnen en faciliteiten.

5.6 IMPLEMENTATIE IN HET PAPIEREN SYSTEEM

Het kan de lezer voorkomen dat de beleidsvormen voor het geheugenbeheer, die in dit hoofdstuk besproken zijn, nogal *ad hoc* lijken en dat de keuze van een algoritme in een bepaald geval nogal arbitrair is. Tot op zekere hoogte is dat waar; veel van de beleidsvormen vinden hun basis op gezond verstand en ervaring, in sommige gevallen gesteund door analytische of statistische rechtvaardiging. Een krachtig hulpmiddel bij de keuze van een algoritme is simulatie; in het ideale geval zou iedere mogelijkheid getest moeten worden met een simulatie van de samenstelling van opdrachten die verwacht wordt. Er zijn een paar studies gemeld door Knuth (1968) en er is een uitgebreide bibliografie verzorgd door Denning (1970).

Voor de implementatie van de mechanismen uit sectie 5.3 in ons papieren besturingssysteem voegen we aan de vluchtige omgeving van elk proces het volgende toe:

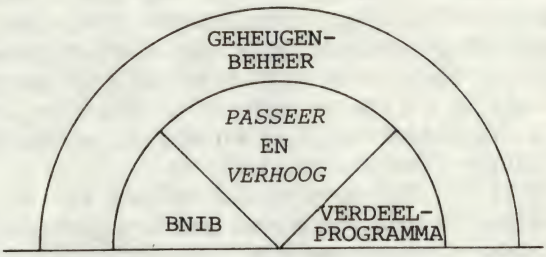
(1) een kopie van de inhoud van de basis- en grensregisters, of
(2) een wijzer naar zijn segmenttabel, of

(3) een wijzer naar zijn paginatabel,

dit al naar gelang de architectuur van onze machine.

Als we een systeem zonder paginaverdeling hebben, voeren we de hele lijst in als een gegevensstructuur waarnaar verwezen wordt vanuit de centrale tabel. De laag met het geheugenbeheer van ons systeem bestaat uit code voor het implementeren van de beleidsvormen die in sectie 5.4 besproken zijn.

Ons systeem heeft nu het stadium bereikt dat in figuur 5.10 is weergegeven.



Figuur 5.10 *De huidige stand van het papieren besturingssysteem*

6 Invoer en uitvoer (Input/Output, I/O)

Op dit punt in de ontwikkeling van ons papieren besturingssysteem hebben we voor een situatie gezorgd waarin processen kunnen bestaan en waarin er geheugenruimte kan worden toegewezen voor het bevatten van de bijbehorende programma's en gegevens. We richten onze aandacht nu op het middel waarmee de processen met de buitenwereld communiceren, dat wil zeggen op de mechanismen voor de in- en uitvoer van informatie.

Van oudsher wordt I/O beschouwd als een van de meer vervelende problemen van het ontwerp van besturingssystemen, dit omdat het een deel is waarin zaken moeilijk in hun algemeenheid gesteld kunnen worden en waarin ad hoc methoden de overhand hebben. De reden hiervoor is de grote variëteit aan toegepaste randapparatuur; een bepaalde configuratie kan randapparaten bevatten die onderling sterk verschillen voor wat hun karakteristieken en werkwijze betreft. In het bijzonder kunnen apparaten op een van de volgende punten verschillen.

(1) Snelheid

Er kan een verschil in gegevensoverdrachtsnelheid zijn dat in de orde van grootte van veelvouden ligt. Zo heeft een magnetische schijf bijvoorbeeld een overdrachtsnelheid van 10^6 tekens per seconde, tegenover het toetsenbord van een invoerstation dat maar een snelheid van een paar tekens per seconde haalt (afhankelijk van de typist!).

(2) De eenheid waarin de overdracht plaatsvindt

Gegevens kunnen in karakter-, woord-, byte-, blok- of record-eenheden overgebracht worden, al naar gelang van de gebruikte randapparatuur.

(3) De manier waarop gegevens weergegeven worden

Een gegeven kan op verschillende manieren op verschillende I/O media weergegeven worden. Zelfs binnen één medium, zoals de magnetische band, kunnen verschillende codes gebruikt worden.

(4) Toegestane handelingen

Randapparatuur verschilt onderling in de soort handelingen die ze kan uitvoeren. Een duidelijk voorbeeld hiervan is natuurlijk het verschil tussen in- en uitvoer; een ander voorbeeld is het kunnen terugdraaien van magnetische band, wat bij het papier van een regeldrukker niet mogelijk is.

(5) Foutomstandigheden

Het niet tot stand komen van een gegevensoverdracht kan verschillende oorzaken hebben zoals: een pariteitsfout, de beschadiging van een kaart of een fout bij een controletelling, afhankelijk van de gebruikte randapparatuur.

Het is duidelijk dat de diversiteit die we hierboven met voorbeelden hebben laten zien moeilijk op één manier af te handelen is. We zullen in dit hoofdstuk echter een kader voor een I/O systeem proberen te scheppen, waarin de apparatuur-afhankelijke eigenschappen zoveel mogelijk apart gehouden worden en waarin een zekere mate van uniformiteit bereikt wordt. We beginnen met het in ogenschouw nemen van enkele ontwerpdoelen en de gevolgen die daaraan vastzitten.

6.1 ONTWERPDOELEN EN DE GEVOLGEN DAARVAN

(1) Onafhankelijkheid van de tekencode

Het is vanzelfsprekend ongewenst dat de programmeur, voor het schrijven van zijn programma, tot in detail kennis moet hebben van de tekencodes die door de diverse randapparaten gebruikt worden. Het I/O systeem moet de verantwoordelijkheid nemen voor de herkenning van de verschillende tekencodes en voor het presenteren van gegevens in een standaardvorm aan de gebruikersprogramma's.

(2) Apparaat-onafhankelijkheid

Er zitten twee aspecten aan de onafhankelijkheid van apparaten. Ten eerste moet een programma onafhankelijk zijn van een specifiek apparaat of van een bepaald type waaraan het verbonden is. Het mag, bijvoorbeeld, niet uitmaken op welk mangeetbandapparaat een bepaalde band gelegd wordt, of welke regeldrukker er voor de uitvoer van een programma gebruikt wordt. Dit soort apparaat-onafhankelijkheid zorgt ervoor dat een programma niet fout loopt vanwege

het simpele feit dat een bepaald apparaat stuk is of ergens anders mee bezig is. Het geeft het besturingssysteem de vrijheid een apparaat van het juiste type toe te wijzen, al naar gelang van de totale beschikbaarheid op dat moment.

Ten tweede - en dat stelt meer eisen - is het wenselijk dat programma's zoveel mogelijk onafhankelijk zijn van het soort apparatuur dat voor hun I/O gebruikt wordt. Het is duidelijk dat dit soort apparatuur-onafhankelijkheid niet zover doorgevoerd kan worden dat het uitvoer naar een kaartlezer gaat sturen; wat we bedoelen is dat er slechts minimale wijzigingen aan een opdracht nodig zouden moeten zijn als deze zijn gegevens van schijf in plaats van van magnetische band moet ontvangen.

(3) Efficiëntie

Omdat I/O handelingen vaak de 'bottleneck' van een computersysteem zijn, is het wenselijk dat zij zo efficiënt mogelijk uitgevoerd worden.

(4) Eenvormige behandeling van apparaten

In het belang van de eenvoud en de vrijheid van fouten is het wenselijk dat alle apparaten op dezelfde manier bestuurd worden. Vanwege al eerder genoemde redenen kan dit in de praktijk moeilijk zijn.

Een paar dingen die uit deze doelstellingen voortvloeien zijn onmiddellijk duidelijk. Ten eerste impliceert onafhankelijkheid van tekencode het bestaan van een uniforme interne weergave van alle tekens. Deze weergave wordt de *interne tekencode* genoemd en er moeten vertalingsmechanismen gekoppeld worden aan elk randapparaat om voor de juiste omzetting van de in- en uitvoer te zorgen. Voor randapparatuur die een diversiteit aan tekencodes aan kan, moet er in een apart vertalingsmechanisme voorzien worden voor elk van de ondersteunde codes. De vertaling vindt onmiddellijk bij invoer en onmiddellijk vóór uitvoer plaats, met als resultaat dat alleen die processen die direct verbonden zijn met het randapparaat zich nog van iets anders dan de standaard interne code bewust moeten zijn. Het vertaalmechanisme zal in het algemeen een tabel zijn, of soms een klein programmastukje.

Ten tweede houdt de onafhankelijkheid van randapparatuur in dat programma's geen handelingen moeten uitvoeren op werkelijke apparaten, maar op virtuele apparaten, met verschillende namen als *stromen* (Atlas), *files* (MULTICS) of *datasets* (IBM). De stromen worden in een programma gebruikt zonder referentie naar de fysieke apparaten; de programmeur verwijst in- en uitvoer slechts naar bepaalde stromen. De koppeling van stromen aan werkelijke apparaten wordt meestal door het besturingssysteem gemaakt, op grond

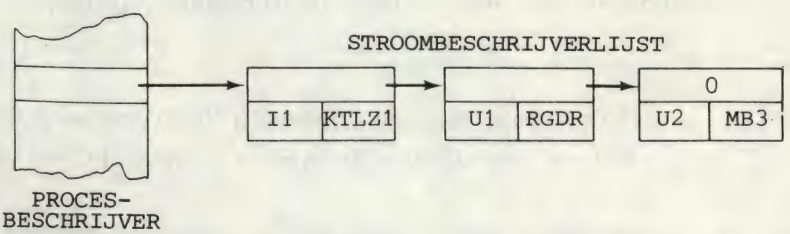
van informatie die door de gebruiker in de opdrachtbeschrijving gegeven is. We zullen opdrachtbeschrijvingen in hoofdstuk 11 in detail bespreken; voor het doel dat we nu voor ogen hebben is het voldoende te zeggen, dat de opdrachtbeschrijving een verzameling door de gebruiker gegeven informatie is, die meestal aan het begin van de opdracht verstrekt wordt en die het besturingssysteem helpt bij de beslissing hoe en wanneer de opdracht uitgevoerd moet worden. Een typisch voorbeeld van een taakbeschrijvingsopdracht voor het koppelen van een stroom aan een randapparaat is

UITVOER 1 = RGDR

wat betekent dat de uitvoerstroom 1 een regeldrukker moet zijn. De onafhankelijkheid van een specifieke regeldrukker wordt gegarandeerd door de vrijheid van het besturingssysteem om de stroom te koppelen aan elke willekeurig beschikbare regeldrukker; apparatuuronafhankelijkheid wordt bereikt door een vanzelfsprekende verandering aan de taakomschrijving (zo kan bijvoorbeeld uitvoer op magnetische band verkregen worden door het veranderen van RGDR in MB).

De overeenkomst tussen stromen en apparaten voor een bepaald proces kan opgeslagen worden in een *stroombeschrijver*-lijst waarnaar verwezen wordt door zijn procesbeschrijver (zie figuur 6.1); de toewijzing van een specifiek apparaat van een bepaald type wordt gedaan als het proces voor het eerst gebruik maakt van de bijbehorende stroom. We zeggen dan dat het proces hier de stroom *opent*; de stroom wordt *gesloten*, wat aangeeft dat er geen verder gebruik meer van gemaakt gaat worden, òf expliciet door het proces, òf impliciet als het proces beëindigd wordt. Aan het proces dat in figuur 6.1 aangegeven wordt, is kaartlezer 1 toegewezen als invoerstroom; magneetbandapparaat 3 is aangewezen als uitvoerstroom 2, en invoerstroom 1 die nog niet geopend is moet een regeldrukker worden.

Een derde gevolg van onze ontwerpdoelstellingen is, dat het I/O systeem zodanig gebouwd moet worden, dat de karakteristieken van apparaten duidelijk aan de apparaten zelf gekoppeld zijn en niet aan de routines die ze besturen (de *apparatuur-besturingsroutines*).



Figuur 6.1 *Apparatuur- en stroominformatie voor een proces*

Het is op deze manier bij apparatuur-besturingsroutines mogelijk dat zij onderling grote overeenkomsten vertonen en dat hun verschillen in handelwijze alleen maar voortkomen uit de parametrische informatie die verkregen wordt uit de karakteristieken van het specifiek betrokken apparaat. De noodzakelijke scheiding van de karakteristieken van het apparaat kan bereikt worden door ze te coderen in een *apparaatbeschrijver* die aan elk apparaat gekoppeld is, en door het gebruik van de apparaatbeschrijver als een informatiebron voor de apparaat-besturingsroutine. De karakteristieke informatie die op deze manier over een apparaat in zijn beschrijver opgeslagen kan worden, is:

(1) de apparaat-identificatie;
(2) de instructies die het apparaat besturen;
(3) wijzers naar omzettabellen voor tekens;
(4) de huidige status: of het apparaat bezet, vrij of kapot is;
(5) het huidige gebruikersproces: een wijzer naar de procesbeschrijver van het eventueel aanwezige proces, dat op het ogenblik het apparaat gebruikt.

Alle procesbeschrijvers kunnen gebundeld worden in een *apparaatstructuur* waarnaar verwezen wordt vanuit de centrale tabel.

6.2 DE I/O PROCEDURES

In de vorige sectie hebben we een aanzienlijke vooruitgang geboekt ten aanzien van het eenvormig behandelen van apparaten, door het afzonderen van alle apparaat-afhankelijke eigenschappen en door deze in de apparaatbeschrijver te plaatsen. We zijn nu in een situatie, waarin we moeten gaan bekijken hoe het besturingssysteem een I/O verzoek van een gebruikersproces afhandelt.

Een typerend verzoek van een proces zal het aanroepen van het besturingssysteem zijn in de algemene vorm:

DOE IO (*stroom, manier, hoeveelheid, bestemming, seinpaal*)

waarin

DOE IO	de naam is van een systeem *I/O procedure*
stroom	het nummer is van de stroom waarop de I/O moet plaatsvinden
manier	aangeeft welke handeling er gevraagd wordt, zoals gegevensoverdrachten of terugspoelen; het kan ook de tekencode die gebruikt wordt aangeven, als dat van toepassing is.

hoeveelheid	de hoeveelheid gegevens is die overgebracht moet worden, als er gegevens overgebracht moeten worden
bestemming	(of bron) de plaats is waarnaar (of waarvandaan) de overdracht is, als die plaatsvindt
seinpaal	het adres is van de seinpaal *verzoek verwerkt*, waarop een verhooghandeling moet worden uitgevoerd als de I/O handeling voltooid is

De DOE IO procedure is 're-entrant' (opnieuw toegankelijk voor een volgende aanroep), zodat hij door meerdere processen tegelijk gebruikt kan worden. Zijn functie is het omzetten van het stroomnummer naar het geschikte fysieke apparaat, het controleren van de samenhang tussen de parameters die eraan gegeven worden, en het starten van de afhandeling van het verzoek.

De eerste van deze handelingen is rechttoe-rechtaan. Welk apparaat overeenkomt met de aangegeven stroom wordt bepaald aan de hand van de informatie, die in de stroombeschrijvingslijst van het vragende proces geplaatst is op het moment dat het geopend werd (zie figuur 6.1). Op het moment dat bekend is welk apparaat gebruikt gaat worden, kan er vergeleken worden of de parameters van het verzoek passen bij de informatie die in de apparaatbeschrijver is opgeslagen. Als er een fout ontdekt wordt, kan er teruggegaan worden naar de aanroeper. Een controle die in het bijzonder uitgevoerd kan worden, is die waarbij bepaald wordt of het apparaat op de gewenste manier kan werken; een andere is of de grootte en bestemming van de gegevensoverdracht overeenkomen met de werkwijze. Bij apparaten die maar één teken tegelijk kunnen overbrengen, moet de aangegeven hoeveelheid over te dragen informatie 1 zijn en moet de bestemming òf een register òf een geheugenplaats zijn; bij apparaten die gegevensblokken direct naar het geheugen overbrengen, moet de aangegeven grootte gelijk zijn aan de blokgrootte (vast of variabel, afhankelijk van het apparaat) en de bestemming moet de geheugenplaats zijn waarnaar de overdracht moet beginnen.

Als de controles voltooid zijn, stelt de I/O procedure de parameters van het verzoek samen tot een *I/O verzoekblok* (IOVB), dat toegevoegd wordt aan een wachtrij van dergelijke blokken, waarin eventueel al andere verzoeken voor het gebruik van hetzelfde apparaat weergegeven staan. Deze andere verzoeken kunnen van hetzelfde proces of, bij het gedeeld gebruik van een apparaat, zoals een schijf, van andere processen af komen. De *apparaat-verzoek-wachtrij* is gekoppeld aan de beschrijver van het betrokken apparaat (zie figuur 6.2) en wordt beheerd door een separaat proces dat *apparaatbesturing (device-handler)* heet en dat we in de volgende sectie zullen beschrijven. De I/O procedure geeft de apparaatbesturing bericht dat het een verzoek in de wachtrij geplaatst heeft, door het uitvoeren van een verhooghandeling op een seinpaal *verzoek wachtend* die bij het betrokken apparaat hoort en die opgeslagen ligt in de apparaatbeschrijver. Als de I/O handeling voltooid

is, dan meldt de apparaatbesturing dit aan het gebruikersproces op een vergelijkbare manier. Dit gebeurt met behulp van de seinpaal *verzoek verwerkt*, waarvan het adres een parameter van de I/O procedure was en die aan de apparaatbesturing werd doorgegeven als een element van het IOVB. Deze seinpaal kan geïnitialiseerd worden òf door het gebruikersproces òf door de I/O procedure wanneer de IOVB gemaakt wordt.

De gehele I/O routine is:

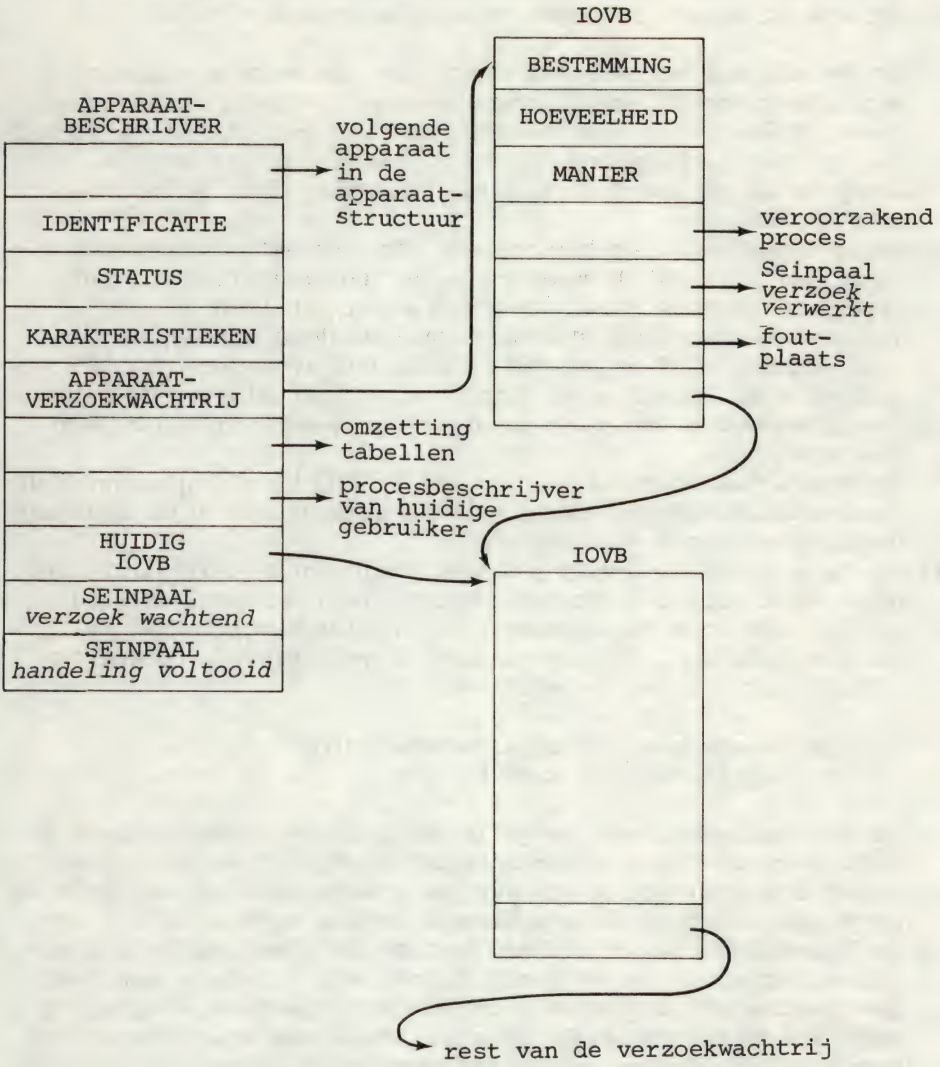
```
procedure DOE IO (stroom,manier,hoeveelheid,bestemming,
                  seinpaal)
begin zoek apparaat op in procesbeschrijver;
      vergelijk parameters met karakteristieken van apparaat;
      als fout dan foutuitgang;
      stel IOVB samen;
      voeg IOVB toe aan apparaat-verzoekwachtrij;
      verhoog (verzoek wachtend)
einde;
```

6.3 DE APPARAATBESTUURDERS

Zoals in de vorige sectie opgemerkt werd, is een apparaatbesturing of apparaatbestuurder een proces dat verantwoordelijk is voor de afhandeling van de verzoeken, die in een apparaat-verzoekwachtrij staan, en voor het melden aan het veroorzakende proces dat de afhandeling voltooid is. Er is een aparte bestuurder voor ieder apparaat, maar omdat alle bestuurders op een vergelijkbare manier werken, kunnen zij gebruik maken van deelbare programma's. Eventuele gedragsverschillen tussen de bestuurders worden afgeleid uit de karakteristieken die in de beschrijvers van de afzonderlijke apparaten zijn opgeslagen.

Een apparaatbestuurder werkt in een continue cyclus, waarin het een IOVB uit de verzoekwachtrij verwijdert en waarin deze dat mededeelt aan het proces dat het verzoek had veroorzaakt. De complete cyclus voor een invoerhandeling is:

```
herhaal onbeperkt
begin passeer (verzoek wachtend)
      haal een IOVB uit de verzoekwachtrij;
      destilleer nadere gegevens uit het verzoek;
      start de I/O handeling;
      passeer (handeling voltooid);
      als fout dan plaats foutinformatie;
      vertaal teken(s) indien nodig;
      breng gegevens over naar bestemming;
      verhoog (verzoek afgehandeld);
      verwijder IOVB
einde;
```



Figuur 6.2 *Apparaatbeschrijver en apparaat-verzoekwachtrij*

De volgende opmerkingen die met betrekking op figuur 6.2 gelezen moeten worden, kunnen de details verduidelijken.

(1) Op de seinpaal *verzoek wachtend*, die zich in de apparaatbeschrijver bevindt, wordt iedere keer een verhooghandeling uitgevoerd door de I/O procedures, als het een IOVB in de verzoekwachtrij voor dit apparaat plaatst. Als de wachtrij leeg is, is de seinpaal nul en wordt de apparaatbesturing tijdelijk buiten werking gesteld.
(2) De apparaatbesturing kan volgens elk gewenst prioriteitsalgoritme een IOVB uit de wachtrij halen. Een algoritme van het type 'wie het eerst komt wordt het eerst geholpen' is meestal voldoende, maar de bestuurder zou beïnvloed kunnen worden door prioriteitsinformatie die door de I/O procedure in de IOVB geplaatst is. Bij een schijf kan de apparaatbesturing de verzoeken afwerken in een volgorde die de verplaatsing van de kop minimaliseert.
(3) De instructies voor het starten van de I/O handeling kunnen uit de apparaatkarakteristieken gehaald worden, die in de apparaatbeschrijver liggen opgeslagen.
(4) Op de seinpaal *handeling voltooid* wordt een verhooghandeling uitgevoerd door de interruptroutine, nadat er een interrupt is veroorzaakt voor dit apparaat. De programmaschets van de interruptroutine, die door de basis niveau ingreep besturing (BNIB) wordt binnengegaan, is:

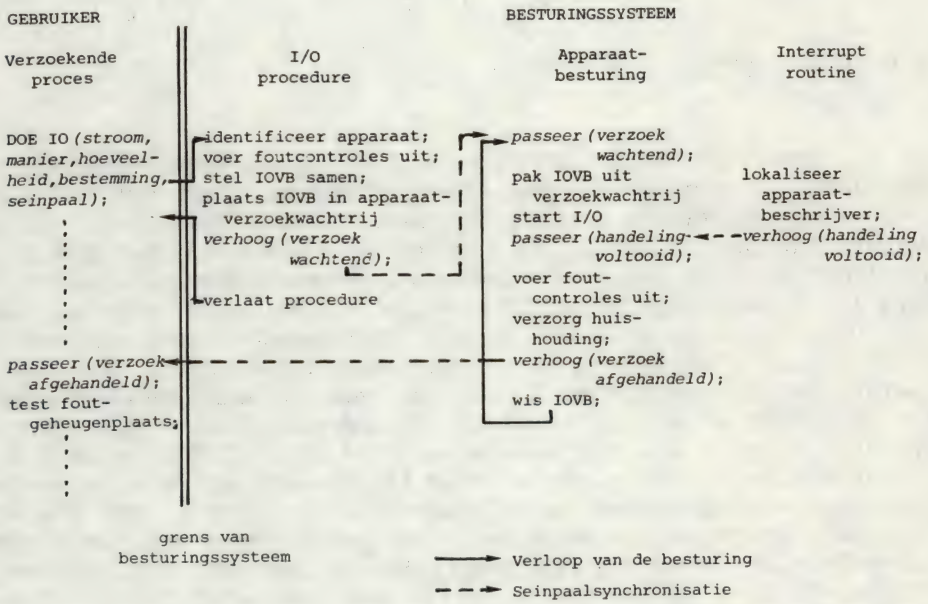
```
stel plaats vast van apparaatbeschrijver;
verhoog (handeling voltooid);
```

De apparaatbeschrijver bergt de seinpaal *handeling voltooid* in zich. Merk op dat de interruptroutine erg kort is; de gehele huishouding wordt gedaan door de apparaatbestuurder die in de gebruikerstoestand als een normaal proces werkt.
(5) De foutcontrole wordt gedaan door het bekijken van de status van het apparaat als de handeling voltooid is. Als er een fout voorgekomen is, zoals een pariteitsfout, kaartbeschadiging, of zelfs een fout veroorzaakt door het einde van een bestand, dan wordt informatie daarover opgeslagen in een *foutgeheugenplaats*, waarvan het adres opgenomen is in de IOVB door de I/O procedure.
(6) De tekenvertaling wordt gemaakt in overeenstemming met de manier die door de IOVB wordt aangegeven en aan de hand van de tabellen waarnaar door de apparaatbeschrijver verwezen wordt.
(7) De hierboven weergegeven cyclus is die voor een invoerhandeling. In het geval van een uitvoerhandeling vindt het halen van gegevens uit hun bron en eventuele tekenvertaling plaats vóór de handeling in plaats van erna.
(8) Het adres van de seinpaal *verzoek afgehandeld* wordt aan de apparaatbestuurder doorgegeven als een deel van de IOVB.

Het wordt als een parameter van de I/O procedure geleverd door het proces dat om I/O verzoekt.

De synchronisatie en het verloop van de besturing tussen het proces, dat om de I/O verzoekt, en de geschikte apparaatbestuurder en routine zijn in figuur 6.3 samengevat. Doorgetrokken pijlen geven de overdracht van besturing aan; gestippelde pijlen geven de synchronisatie met behulp van seinpalen weer. Het is belangrijk op te merken dat in het weergegeven schema het verzoekende proces asynchroon verloopt met de apparaatbesturing, zodat het andere berekeningen kan uitvoeren of andere I/O verzoeken kan doen terwijl het eerste verzoek nog verzorgd wordt. Het verzoekende proces hoeft alleen maar onderbroken te worden bij gelegenheden waarbij de I/O behandeling nog niet voltooid is wanneer het de behandeling *passeer (verzoek afgehandeld)* uitvoert.

Het nadeel van deze werkwijze is dat het verzoekende proces de verantwoordelijkheid moet nemen voor het synchroniseren van zijn eigen activiteiten met die van de apparaatbesturing. Het moet bijvoorbeeld niet proberen invoergegevens te gebruiken die nog niet gegeven zijn. Dit betekent dat de schrijver van het proces zich ervan bewust moet zijn dat I/O handelingen niet in één ogenblik plaatsvinden en dat ze, met behulp van een juist gebruik van de seinpaal *verzoek afgehandeld* in staat moeten zijn de gewenste synchronisatie te bereiken.



Figuur 6.3 *Schets van het I/O systeem*

Een alternatieve werkwijze is het leggen van de verantwoordelijkheid voor de synchronisatie binnen de I/O procedure, die een deel van het besturingssysteem is. Dit kan door de seinpaal *verzoek afgehandeld* plaatsgebonden te maken aan de I/O procedure (dat wil zeggen dat zijn adres niet langer gegeven hoeft te worden door het verzoekende proces) en door het, onmiddellijk voor het verlaten van de I/O procedure, invoegen van de handelingen

passeer (verzoek afgehandeld);
test foutgeheugenplaats;

De vertraging die verbonden is aan een I/O handeling is nu in het besturingssysteem verborgen; voor wat het verzoekende proces aangaat is de handeling ogenblikkelijk, in die zin dat als gevolg wordt gegeven aan de volgende instructie na het verzoek, ervan uit kan worden gegaan dat de handeling compleet is. Anders gezegd, het verzoekende proces loopt op een virtuele machine, waarin I/O handelingen bereikt worden door een enkele ogenblikkelijke instructie.

De twee hierboven besproken werkwijzen kunnen als volgt worden samengevat. In de eerste heeft de gebruiker de vrijheid parallel aan de I/O handeling verder te gaan, maar hij heeft de verantwoordelijkheid na te gaan of die voltooid is; in de tweede wordt hem die verantwoordelijkheid ontnomen, maar hij verliest dan ook de vrijheid.

6.4 BUFFERING

De beschrijving van de I/O procedure en de apparaatbesturing, die hierboven gegeven is, gaat ervan uit dat alle gegevensoverdracht ongebufferd is. Dat wil zeggen dat ieder I/O verzoek van een proces fysieke overdracht veroorzaakt van of naar het geschikte randapparaat. Als een proces herhaalde overdrachten uitvoert op dezelfde stroom, dan zal het gedurende de overdracht herhaaldelijk moeten wachten (op *passeer (verzoek afgehandeld)*). Om veel extra werk te voorkomen bij het schakelen tussen processen is het soms handig de I/O overdracht uit te voeren vóórdat erom verzocht wordt. Op die manier wordt verzekerd dat de gegevens beschikbaar zijn wanneer ze nodig zijn. Deze techniek staat als *buffering* bekend.

Invoeroverdrachten worden door het besturingssysteem gestuurd naar een geheugengebied dat invoerbuffer heet; het gebruikersproces haalt zijn gegevens uit de buffer en wordt alleen dan gedwongen te wachten als de buffer leeg raakt. Als dat gebeurt, vult het besturingssysteem de buffer weer en het proces gaat weer verder. Overeenkomstig hiermee worden uitvoeroverdrachten van een proces naar een uitvoerbuffer gestuurd en het besturingssysteem maakt de hele buffer leeg als die vol is. Het proces hoeft alleen te wachten wanneer

het iets probeert uit te voeren voordat het systeem de buffer heeft leeggemaakt.

De techniek kan verfijnd worden door het gebruik van twee buffers in plaats van één, de zogenaamde *dubbele buffering*. Een proces brengt nu gegevens over naar (of van) een buffer terwijl het besturingssysteem de andere buffer leeg (of vol) maakt. Dit zorgt er met zekerheid voor dat processen alleen maar wachten als beide buffers vol (of uitgeput) zijn voordat het besturingssysteem actie ondernomen heeft. Dit komt alleen voor als het proces een plotselinge uitbarsting van I/O geeft en in dat geval kan het probleem opgelost worden door het toevoegen van nog meer buffers (*meervoudige buffering*). Bij gevallen waarin een proces voortdurend een hogere I/O snelheid vraagt dan die waarmee de apparaten kunnen werken, kan natuurlijk geen enkele hoeveelheid buffering helpen. Het nut van buffering is beperkt tot het uitvlakken van pieken in de I/O behoefte, in situaties waarin de *gemiddelde* behoefte niet groter is dan de apparaten verwerken kunnen. Algemeen kan men stellen: hoe groter de pieken des te groter het aantal buffers dat nodig is voor de egalisatie.

Of een stroom al dan niet gebufferd moet worden kan aangegeven worden door een daarvoor geschikte opdracht in de taakomschrijving; in het geval van buffering kan de gebruiker aangeven hoeveel buffers er gebruikt moeten worden, of hij kan het systeem een van tevoren ingestelde waarde, gewoonlijk twee, laten gebruiken. Het besturingssysteem wijst de ruimte voor de buffers toe wanneer de stroom wordt geopend en slaat de bufferadressen op in de stroombeschrijver.

Een iets andere I/O procedure is nodig voor het mogelijk maken van een gebufferde handeling. De nieuwe procedure verwerkt een invoerverzoek door het onttrekken van het volgende onderdeel aan de juiste buffer en door het direct door te geven aan het vragende proces. Alleen als de buffer leeg raakt, veroorzaakt de procedure een IOVB en geeft het een signaal aan de apparaatbesturing voor het geven van meer invoer. Als de stroom geopend wordt, genereert de I/O procedure genoeg IOVBs voor het vullen van alle buffers. De apparaatbesturing werkt zoals we eerder beschreven hebben, hij initieert een gegevensoverdracht naar de buffer waarvan de geheugenplaats aangegeven wordt in de IOVB. Vergelijkbare opmerkingen gelden, *mutatis mutandis*, voor de verzoeken om uitvoer. In beide gevallen vormen de I/O procedure en de apparaatbesturing een variatie op het producent/verbruikerspaar dat in hoofdstuk 3 is beschreven.

De I/O procedure voor gebufferde handelingen kan worden aangeroepen uit een gebruikersproces door middel van een opdracht met de algemene vorm

DOE BUFIO (*stroom,manier,bestemming*)

waarin *stroom*, *manier* en *bestemming* hetzelfde zijn als beschreven voor de procedure DOE IO in sectie 6.2. De hoeveelheid overgebrachte

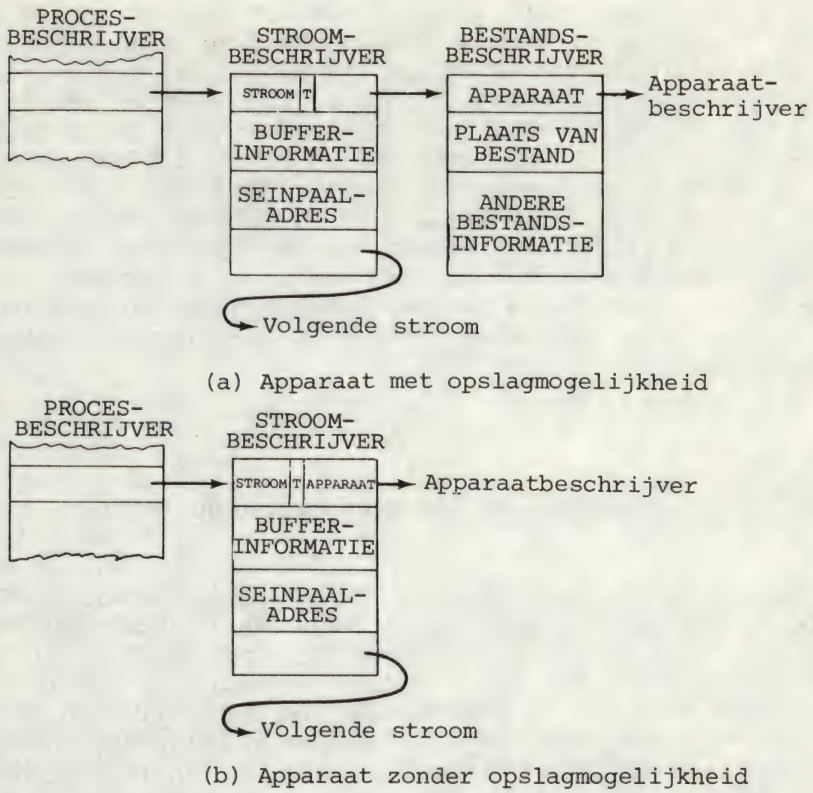
informatie zal een enkel deel zijn. Merk op dat, omdat vertragingen die voortkomen uit volle of lege buffers verborgen zijn in de I/O procedure, het niet nodig is een seinpaaladres als één van de parameters mee te geven. Het type buffering (indien gebruikt), de adressen van de buffers en de seinpalen die door de I/O procedures gebruikt gaan worden, zijn allemaal toegankelijk vanuit de stroombeschrijvingslijst van het proces dat de aanvraag doet (zie figuur 6.4b). Het eerste element uit deze informatie kan gebruikt worden om vast te stellen welke I/O procedure aangeroepen moet worden voor het uitvoeren van een overdracht.

6.5 OPSLAGAPPARATUUR

In de voorafgaande bespreking hebben we stilzwijgend aangenomen dat de naam van een randapparaat voldoende informatie vormt voor het vaststellen van de externe bron of bestemming van een bepaalde overdracht. Dit gaat op bij randapparaten die sequentieel werken, zodat er geen twijfel bestaat over het gebied in het externe medium waarnaar of waaruit de gegevensoverdracht gericht is. Zo kan bijvoorbeeld de uitlezing van een toetsenbord alleen het volgende teken dat getypt wordt lezen, terwijl een regeldrukker alleen maar op de huidige lijn kan drukken. In sommige gevallen is het mogelijk het medium een bepaalde hoeveelheid (bijvoorbeeld naar een nieuwe pagina) vooruit te laten gaan, maar er is geen sprake van een mogelijkheid om een overdracht naar wens op elk deel te richten. Andere apparatuur, zoals schijf- en trommelstations, die op een willekeurig toegankelijke manier werken, voorzien in een mogelijkheid voor het kiezen van een bepaald deel van het medium (schijf of trommel), waarop de overdracht uitgevoerd moet worden. In die gevallen is het niet voldoende de naam van het betrokken apparaat te noemen; het is ook nodig dat er gespecificeerd wordt welk gegevensgebied op het medium gebruikt moet worden.

Ieder gegevensgebied, dat op een dergelijk medium kan bestaan, wordt *bestand* (Engels: *file*) genoemd en heeft meestal een arbitraire grootte die gedefinieerd wordt wanneer het wordt aangemaakt of bijgewerkt. Een apparaat dat bestanden ondersteunt wordt *opslagapparaat* genoemd. In het volgende hoofdstuk zullen we meer over de organisatie van bestanden vertellen; voor het ogenblik zullen we aannemen dat ieder bestand een unieke naam heeft, die door het besturingssysteem gebruikt kan worden voor het vaststellen van de plaats van het bestand op het medium. Een inhoudsopgave van de bestandsnamen, met hun bijbehorende plaatsen, wordt voor dit doel door het besturingssysteem bijgehouden.

Als een gegevensstroom naar of van een apparaat met bestandsstructuur gestuurd moet worden, dan wordt die stroom in de taakomschrijving gekoppeld aan de naam van een bepaald bestand in plaats



Figuur 6.4 *Stroominformatie voor apparaten met en zonder opslagmogelijkheid (T = soort apparaat)*

van aan de naam van het apparaat. Een typerende taakomschrijvingsopdracht zou kunnen zijn:

```
INVOER 1 = 'TESTGEGEVENS'
```

dit geeft aan dat de gegevens op stroom 1 uit een bestand met de naam TESTGEGEVENS moeten komen. Als de stroom geopend wordt, zoekt het besturingssysteem de bestandsnaam op in de inhoudsopgave om zo het betreffende apparaat en de betreffende plaats te vinden waarop het bestand is opgeslagen. Deze procedure, die bekend staat als het *openen* van het bestand, bevat diverse controles die in het volgende hoofdstuk nader beschreven zullen worden; omdat het een nogal lange handeling kan zijn, is het niet wenselijk deze voor elke gegevensoverdracht te herhalen. Dientengevolge wordt er elke keer dat een bestand geopend wordt een *bestandsbeschrijver* gemaakt voor het opslaan van de gegevens die nodig zijn bij nog komende overdrachten. Deze informatie bevat ondermeer:

het adres van de apparaatbeschrijver van het apparaat waarop het bestand is opgeslagen, de plaats van het bestand op het apparaat, of er in het bestand geschreven mag worden, of dat eruit gelezen mag worden, en nadere gegevens over de interne organisatie van het bestand. Een wijzer naar de bestandsbeschrijver wordt in de juiste stroombeschrijver geplaatst van het proces dat het bestand opent, zoals in figuur 6.4a is weergegeven. De lezer zal aan de hand van een vergelijking tussen figuur 6.4a en 6.4b zien dat de bestandsbeschrijver extra informatie toevoegt aan de koppeling van stromen aan fysieke apparatuur. De I/O procedures, die de koppeling voor elke gegevensoverdracht verzorgen, kunnen in een handomdraai veranderd worden, zodat zij met deze informatie rekening houden bij het samenstellen van een IOVB.

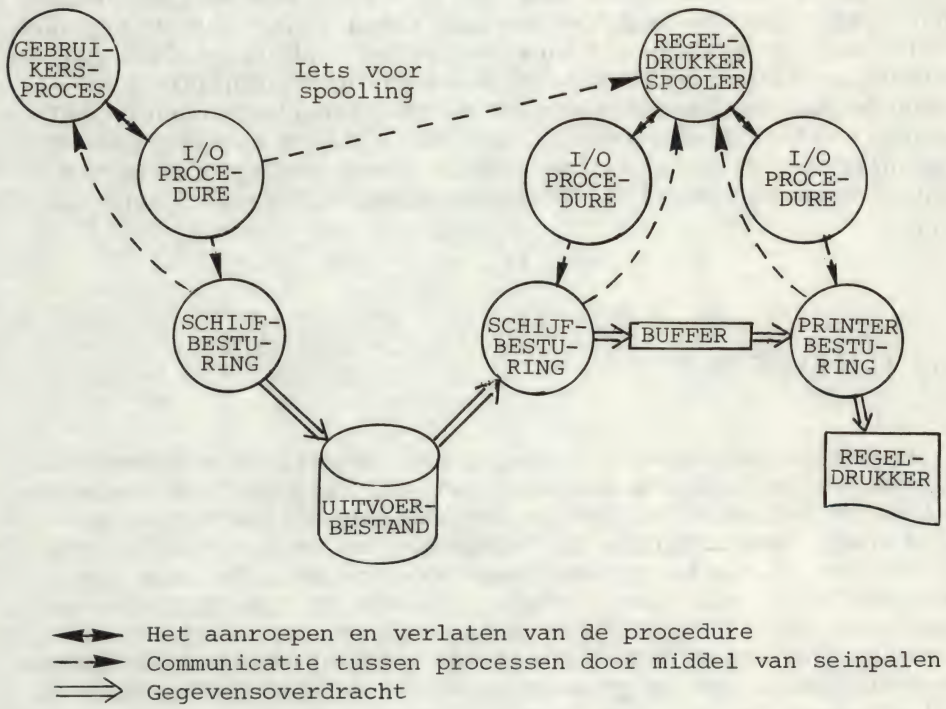
6.6 SPOOLING (Simultaneous Peripheral Operation On Line)

De voorafgaande secties hebben een vanzelfsprekend onderscheid gemaakt tussen deelbare apparatuur, zoals schijfstations, en ondeelbare apparatuur, zoals regeldrukkers, die noodgedwongen aan één proces tegelijk zijn toegewezen. Ondeelbare apparaten zijn die, welke zo werken dat hun toewijzing aan meerdere processen een onontwarbare vermenging van I/O transporten tot gevolg zou hebben. Zoals in sectie 6.1 werd duidelijk gemaakt, vindt de toewijzing van een ondeelbaar apparaat plaats wanneer een proces een stroom opent die eraan gekoppeld is; het apparaat wordt pas vrijgegeven als de stroom gesloten wordt of als het proces afloopt. Processen die gebruik willen maken van een apparaat dat al toegewezen is, moeten wachten totdat het vrijgegeven wordt. Dit betekent dat er, gedurende periodes dat er een grote vraag is, diverse processen opgehouden kunnen worden doordat ze wachten op het gebruik van schaarse apparatuur, terwijl gedurende andere periodes diezelfde apparatuur ongebruikt kan zijn. Voor het spreiden van de last en het terugbrengen van de kans op knelpunten kan er behoefte zijn aan een andere strategie.

De oplossing die door veel systemen gebruikt wordt, is *spooling* van alle I/O bij zwaarbelaste apparaten. Dit betekent dat alle overdracht niet direct naar het apparaat dat aan de stroom gekoppeld is gestuurd wordt, maar dat de I/O procedures de overdracht uitvoeren op het een of andere tussenmedium, meestal een schijf. De verantwoordelijkheid voor het overbrengen van de gegevens tussen de schijf en het gevraagde apparaat, berust bij een separaat proces, een *spooler*, dat bij het apparaat hoort. Laten we als voorbeeld een systeem bekijken waarin alle uitvoer naar de regeldrukker met behulp van spooling gaat. Aan ieder proces, dat een stroom voor de regeldrukker opent, wordt een anoniem bestand op de schijf toegewezen en alle uitvoer op de stroom wordt naar dit bestand geleid

door de I/O procedure. Het bestand werkt in feite als een virtuele regeldrukker. Als de stroom gesloten wordt, wordt het bestand toegevoegd aan een wachtrij met dergelijke bestanden die door andere processen zijn aangemaakt en die allemaal wachten tot zij gedrukt worden. De functie van de regeldrukker-spooler is: het halen van bestanden uit de wachtrij en het zenden ervan naar de regeldrukker. Er wordt natuurlijk aangenomen dat, over een langere tijdsduur, de snelheid van de printer voldoende is voor het verwerken van alle uitvoerbestanden die aangemaakt worden. Een kale structuur voor de spooler is:

```
herhaal onbeperkt
begin   passeer (iets voor spooling);
        haal bestand uit wachtrij;
        open bestand;
        herhaal tot einde van bestand;
        begin DOE IO (parameters voor uitlezen schijf);
              passeer (schijfverzoek afgehandeld);
              DOE IO (parameters voor afdrukeenheiduitvoer);
              passeer (regeldrukkerverzoek afgehandeld);
        einde
einde;
```



Figuur 6.5 *Spooling van uitvoer*

De lezer kan de volgende punten opmerken:

(1) De structuur zal in werkelijkheid veranderd worden om buffering van gegevens tussen de schijf en de afdrukeenheid mogelijk te maken.
(2) Op de seinpaal *iets voor spooling* worden verhooghandelingen uitgevoerd door elk proces dat een regeldrukkerstroom sluit (dat wil zeggen een uitvoerbestand voltooit).
(3) De uitvoerwachtrij hoeft niet verwerkt te worden volgens het principe 'wie het eerst komt wordt het eerst geholpen'; de spooler kan bijvoorbeeld korte bestanden bevoordelen te opzichte van lange.

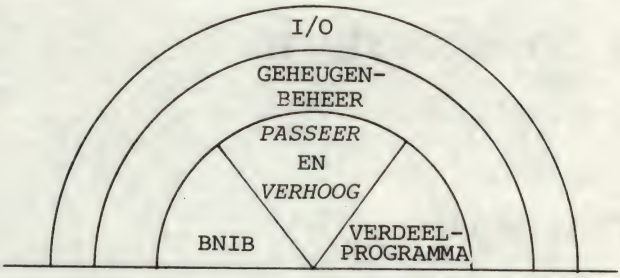
De onderlinge relaties tussen de spooler, de apparaatbestuurders en de processen die de uitvoer aanmaken is weergegeven in figuur 6.5. Een dergelijk diagram, dat gebaseerd is op een bespreking die analoog is aan de bovenstaande, zou voor een invoerspooler getekend kunnen worden.

Samenvattend kunnen we zeggen dat spooling de druk van de aanvragen op intensief gebruikte randapparaten egaliseert. Zoals we in hoofdstuk 8 zullen zien, vermindert het ook de kans op het vastlopen van het systeem door het onoordeelkundig toewijzen van randapparatuur. Een ander voordeel is dat het relatief gemakkelijk te verwezenlijken is meerdere kopieën van dezelfde uitvoer te maken zonder dat de opdracht opnieuw uitgevoerd hoeft te worden. Aan de debetzijde staat de grote hoeveelheid benodigde schijfruimte die nodig is voor het bevatten van de in- en uitvoerwachtrijen en het drukke verkeer op het schijfkanaal. Ten slotte is spooling natuurlijk niet werkbaar in een real-time situatie waarin de gegevens direct verwerkt worden omdat de I/O behandeling daar ogenblikkelijk moet zijn.

6.7 TEN SLOTTE

In de voorgaande secties hebben we een I/O systeem geschetst dat voldoet aan de doelstellingen voor code-onafhankelijkheid, apparaat-onafhankelijkheid en eenvormige behandeling van alle randapparatuur. Deze eigenschappen zijn verkregen ten koste van efficiëntie; de I/O procedures en apparaatbestuurders zullen, vanwege hun algemene aard, langzamer werken dan kleine stukjes maatwerk-code voor bepaalde I/O handelingen of apparatuur. Het kader dat we neergezet hebben is echter inhoudelijk goed en kan als basis dienen voor optimalisering. De grootste winst zou in de praktijk komen uit het vervangen van de code voor I/O procedures en apparaatbestuurders, door stukjes programma die specifiek voor een apparaat of handeling zijn. Dat zou natuurlijk wel de uniforme benadering verstoren die we tot stand gebracht hebben.

Het huidige stadium in de constructie van ons papieren besturingssysteem is in figuur 6.6 weergegeven.



Figuur 6.6 *De huidige staat van ons papieren besturingssysteem*

7 Het opslagsysteem

7.1 DOELSTELLINGEN

In hoofdstuk 2 hebben we verteld dat de mogelijkheid van een lange-termijnopslag een wenselijke eigenschap van een besturingssysteem is. De motivatie voor een lange-termijnopslag varieert al naar gelang de aard van het systeem, maar komt meestal voort uit een van de volgende voordelen.

(1) Gekoppelde, direct beschikbare opslag (Engels: on line storage)

Voor bepaalde toepassingen die te maken hebben met het terughalen van informatie (zoals management informatiesystemen), is het nodig dat grote hoeveelheden gegevens zo opgeslagen worden dat ze altijd toegankelijk zijn. Zelfs bij een systeem voor algemeen gebruik, waarbij een direct beschikbaar gegevensbestand niet noodzakelijk hoeft te zijn, is het een groot gemak voor de gebruiker als hij programma's of gegevens in het verwerkingssysteem zelf kan opslaan, in plaats van op het een of andere externe medium zoals ponskaarten. In het bijzonder is het bij meervoudig toegankelijke systemen irreëel te verwachten dat de gebruiker het zonder direct gekoppelde informatie kan stellen, dit omdat het enige I/O apparaat dat hij tot zijn beschikking heeft het invoerstation is waaraan hij zit te typen. Maar weinig gebruikers zouden een systeem tolereren waarbij alle programma's en gegevens bij het begin van elke sessie opnieuw op het invoerstation ingetikt moeten worden. Zelfs bij batchsystemen kan direct gekoppelde informatie-opslag de verwerkingscapaciteit verhogen door het verminderen van de afhankelijkheid van trage randapparatuur.

(2) Gemeenschappelijk gebruik van informatie

Het is bij sommige systemen wenselijk dat gebruikers in staat zijn gemeenschappelijk gebruik te maken van informatie. Zo kunnen de gebruikers van een systeem voor algemene doeleinden bijvoorbeeld elkaars programma's of gegevens willen gebruiken, en in een systeem dat transacties verwerkt kunnen veel afzonderlijke processen

hetzelfde gegevensbestand gebruiken. Bij de meeste systemen voor algemeen gebruik is het wenselijk dat er een *programmabibliotheek* geïnstalleerd wordt met programma's voor het werken met teksten (editors), vertaalprogramma's (compilers), of wetenschappelijk subroutines die algemeen beschikbaar zijn voor de gebruikers van het systeem. Als informatie op deze manier gedeeld moet worden, dan moet die direct toegankelijk opgeslagen worden gedurende lange periodes.

Vanwege economische redenen vindt de lange-termijnopslag plaats op secundaire media zoals schijven, trommels en magnetische banden; het doel van het *opslagsysteem* is het zorgen voor de hulpmiddelen voor het organiseren en benaderen van de gegevens, op een voor de gebruiker handige manier. De manier waarop dat gebeurt, hangt natuurlijk af van de soort gegevens en de toepassing die ze krijgen; er valt te verwachten dat het opslagsysteem voor een transacties verwerkend systeem verschilt van dat voor een procesbesturingssysteem. In dit hoofdstuk zullen we in de geest van dit boek blijven door ons bezig te houden met systemen voor algemeen gebruik, waarbij de opslagsystemen meestal op schijven zijn gebaseerd; de lezer die geïnteresseerd is in andere gebieden wordt verwezen naar de uitgebreide literatuur elders (bijvoorbeeld IFIP, 1969; Judd, 1973; Martin, 1977, 1983).

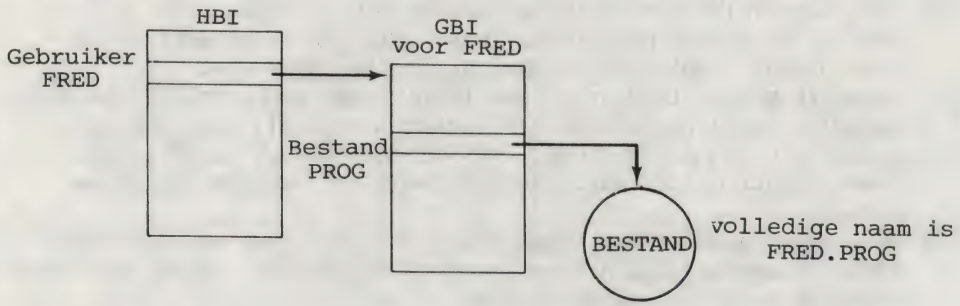
De gebruiker van systemen voor algemeen gebruik rangschikt zijn gegevens in *bestanden* van willekeurige grootte. Elk bestand is zodoende een gegevensverzameling die door de gebruiker als een wenzenlijke eenheid beschouwd wordt; het kan een programma, een set procedures of de resultaten van een experiment zijn. Het bestand is de logische eenheid die door het opslagsysteem opgeslagen en gemanipuleerd wordt. Het medium waarop bestanden opgeslagen worden, is meestal in *blokken* van een vaste lengte (meestal tussen de 512 en 2048 bytes) ingedeeld en het opslagsysteem moet het juiste aantal blokken aan ieder bestand toewijzen. Om bruikbaar te zijn moet het opslagsysteem

(1) het aanmaken en wissen van bestanden toestaan;
(2) toegang tot bestanden toestaan voor het lezen of schrijven;
(3) automatisch de besturing verzorgen van de secundaire geheugenruimte. Waar zijn bestanden in het secundaire geheugen zijn opgeslagen, moet voor de gebruiker niet uitmaken;
(4) verwijzing naar bestanden met behulp van een symbolische naam mogelijk maken. Omdat de gebruiker niet weet, noch wenst te weten wat de fysieke plaats van zijn bestanden is, zou hij aan alleen de naam voldoende hebben voor het verwijzen naar een bestand;
(5) bestanden beschermen tegen storing in het systeem. Gebruikers zullen weigeren zich aan het systeem te binden, tenzij ze van de integriteit ervan overtuigd zijn;
(6) het gemeenschappelijk gebruik van bestanden door samenwerkende gebruikers toestaan, maar het moet bestanden beschermen tegen de benadering door niet geautoriseerde gebruikers.

In de volgende secties zullen we nagaan hoe de bovengenoemde doelstellingen bereikt kunnen worden.

7.2 BESTANDSINDEXEN

Het basisprobleem bij het benaderen van een bestand is het omzetten van een symbolische naam naar een fysieke plaats in het secundaire geheugen. De vertaling wordt bereikt met behulp van een *bestandsindex* of *catalogus*, dat is in principe een tabel die informatie bevat over de verblijfplaats van genoemde bestanden. Aangezien de index het mechanisme is waarlangs het bestand benaderd wordt, is het logisch om hierin een bewakingsmiddel op te nemen tegen niet-geautoriseerde toegang. We zullen dat verder bespreken in de volgende sectie; voor het moment zien we dat een directe beschermingsmaatregel mogelijk is door het verdelen van de index in twee niveaus, zoals in figuur 7.1 weergegeven. In het bovenste deel bevat een *hoofdbestandsindex* (HBI) voor elke gebruiker in het systeem een wijzer naar een *gebruikersbestandsindex* (GBI) voor die gebruiker; in het onderste deel bevat iedere GBI de namen en plaatsen van de bestanden van een enkele gebruiker. Omdat een GBI alleen benaderd kan worden langs de HBI, kan de privacy van de bestanden van een gebruiker verzekerd worden door een eenvoudige identiteitscontrole op het HBI-niveau. Het is zelfs mogelijk dat verschillende gebruikers dezelfde naam gebruiken voor een bestand, zonder dat er verwarring ontstaat, omdat de *volledige naam* van een bestand beschouwd kan worden als de aaneenschakeling van de naam (of het nummer) van de gebruiker en de *individuele naam* van het bestand. Zo is bijvoorbeeld de individuele naam van het bestand uit figuur 7.1 'PROG' en zijn volledige naam is 'FRED.PROG'. Een ander bestand met dezelfde individuele naam, maar dat van gebruiker PAUL is, zou als volledige naam 'PAUL.PROG' hebben. In de praktijk is het niet altijd nodig de



Figuur 7.1 *Indexstructuur met twee niveaus*

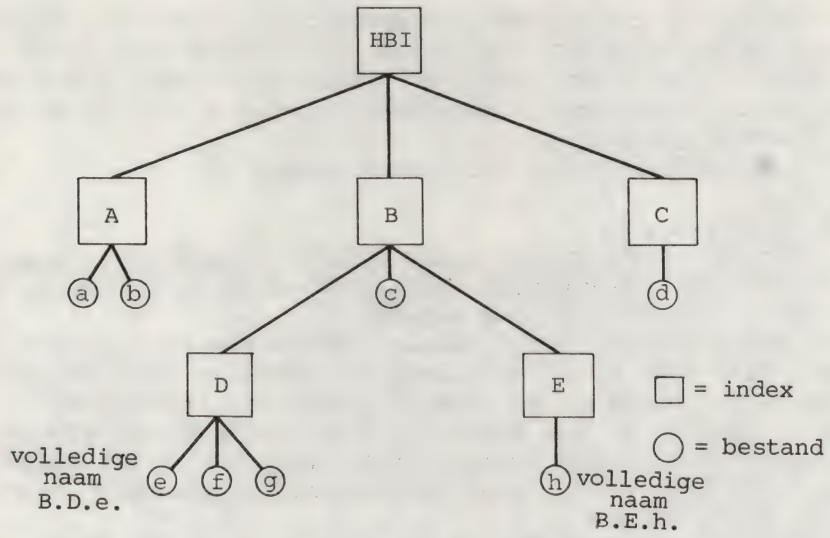
volledige naam te specificeren, omdat het opslagsysteem de identiteit van degene die toegang vraagt als een vooraf ingestelde waarde voor het eerste deel kan gebruiken. Alleen als de gebruiker toegang vraagt tot de bestanden van een andere gebruiker moet hij de volledige naam aangeven.

De informatie in iedere GBI bestaat meestal uit:

(1) De bestandsnaam.
(2) De fysieke plaats van het bestand in het secundaire geheugen. De vorm van zijn ingang zal afhangen van de manier waarop het bestand is opgeslagen (zie sectie 7.4).
(3) Het bestandstype (tekens, binair, verplaatsbaar, enzovoort). Deze informatie wordt voornamelijk bijgehouden voor het gemak van de gebruiker en ook voor het gemak van systeemprogramma's, zoals programma's voor het laden van gegevens of voor editors, die gebruikt kunnen worden voor het werken met het bestand. Voor wat het opslagsysteem zelf aangaat is elk bestand gewoon een rij bits.
(4) Informatie voor de besturing van de toegang (bijvoorbeeld alleen lezen; zie sectie 7.3).
(5) Administratieve informatie (bijvoorbeeld de datum van de laatste keer dat er een aanpassing plaatsvond). Deze informatie wordt gebruikt voor het voorzien in gegevens voor de systeemwerkzaamheden die gedupliceerde kopieën bewaren, als een verzekering tegen hardwarestoringen (zie sectie 7.5).

Veel systemen (bijvoorbeeld het DEC System-10) maken gebruik van de index met twee niveaus zoals hierboven beschreven is. Andere, zoals UNIX, breiden het concept uit tot een structuur met meerdere niveaus, waarbij ingangen wijzers kunnen zijn naar bestanden of naar andere indexen (zie figuur 7.2). Een structuur met meerdere niveaus is nuttig in situaties waarbij de opgeslagen informatie een classificatie heeft in een vertakte vorm, of in situaties waarbij de gebruikers gegroepeerd zijn in hiërarchieën, zoals personen in werkgroepen in afdelingen. In het tweede geval kan er een hiërarchisch beschermingsmechanisme toegepast worden door het gebruiken van steeds stringentere controles als men verder afzakt in de vertakking. Net zoals in het systeem met twee niveaus kunnen met elkaar in botsing komende bestandsnamen voorkomen worden door de volledige naam van het bestand te beschouwen als de aaneenschakeling van zijn individuele naam aan de namen van de indexen op zijn benaderingspad

Het nadeel van een systeem met meerdere niveaus ligt in de lengte van het pad naar een bepaald bestand en in het aantal keren dat de schijf benaderd moet worden voor het volgen van het pad langs de diverse indexen. Dit kan tot op zekere hoogte vereenvoudigd worden door ervan uit te gaan dat opeenvolgende benaderingen van bestanden vaak gericht zijn op bestanden in dezelfde index. Als er eenmaal een weg door de structuur van de vertakking is vastgesteld naar een bepaalde index, dan kan deze index aangewezen



Figuur 7.2 *Structuur van een index met meerdere niveaus*

worden als de *huidige index* en volgende verwijzingen naar een bestand hoeven alleen de individuele bestandsnaam te noemen. Een wisseling van de huidige index wordt door het aanroepen van de volledige bestandsnaam veroorzaakt. Deze techniek wordt in UNIX en MULTICS gebruikt (Daley en Neumann, 1965).

7.3 GEMEENSCHAPPELIJK GEBRUIK EN VEILIGHEID

Het veiligheidsprobleem van bestanden komt direct voort uit de wens ze gemeenschappelijk te kunnen gebruiken. In een situatie waarin er geen gemeenschappelijk gebruik plaatsvindt - de enige persoon die een bestand mag benaderen is de eigenaar zelf - kan de bescherming bereikt worden door het uitvoeren van een identiteits-controle in de HBI zelf. Als bestanden echter gemeenschappelijk gebruikt moeten worden, heeft de eigenaar een middel nodig om aan te geven welke gebruikers toegang tot zijn bestanden mogen krijgen en welke niet. Het is tevens praktisch voor de eigenaar als hij kan aangeven *wat voor* toegang toegestaan moet worden; hij kan wensen dat een aantal van zijn vrienden de mogelijkheid moet hebben zijn bestanden bij te werken, terwijl anderen ze slechts mogen lezen en nog anderen ze slechts mogen laden om ze uit te voeren. We kunnen dit samenvatten door te zeggen dat de eigenaar in staat moet zijn om aan te geven welke *toegangsprivileges* andere gebruikers mogen hebben.

Een bijzonder eenvoudige manier hiervoor is het koppelen van een set toegangsprivileges, die van toepassing zijn voor de verschillende gebruikersklassen, aan ieder bestand. De gebruikersklassen zouden de volgende kunnen zijn:

(1) de eigenaar,
(2) partners,
(3) anderen,

en typische toegangsprivileges zijn:

(1) geen toegang (G),
(2) alleen uitvoeren (U),
(3) alleen lezen (L),
(4) alleen toevoegen (T),
(5) bijwerken (B),
(6) veranderen bescherming (V),
(7) wissen (W).

Elk privilege houdt in het algemeen ook de erboven liggende in. Een duidelijk voorbeeld van een *bestandsbeschermingssleutel*, die bij de aanmaak van een bestand door de eigenaar eraan kan worden toegewezen, zou WTL zijn; dit houdt in dat de eigenaar alles kan doen, de partners kunnen toevoegen en alle anderen kunnen lezen. De beveiligingssleutel voor het bestand zetelt in de GBI, als een deel van de ingang voor dat bepaalde bestand. Merk op dat het in sommige gevallen verstandig kan zijn dat de eigenaar zichzelf maar een laag toegangsprivilege geeft, dit om zichzelf tegen zijn eigen vergissingen te beschermen. Desondanks moet het opslagsysteem de eigenaar altijd toestaan het beschermingsprivilege te veranderen, omdat hij anders geen mogelijkheid zou hebben zijn eigen bestand te wijzigen of te wissen.

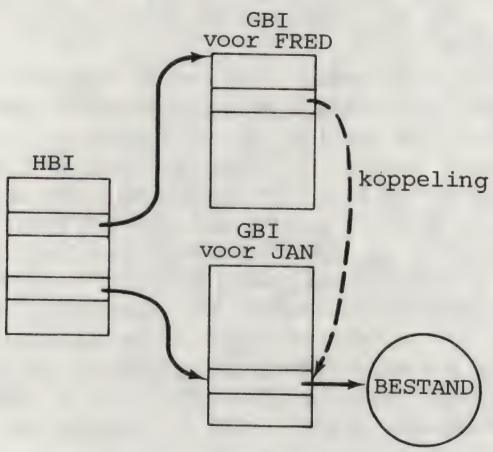
Een gevolg van deze methode is dat de eigenaar in zijn GBI in een lijst moet opgeven wie als partners benoemd zijn. Deze lijst moet voor controle nagelopen worden bij alle toegangen die niet door de eigenaar zelf gedaan worden. In sommige systemen is het mogelijk dit te vermijden door het impliciet definiëren van het partnerschap: de identificatie waardoor de gebruiker aan het systeem bekend is kan zo gestructureerd zijn dat zijn eerste deel het partnerschap en het tweede deel de individuele gebruikersidentiteit aangeeft. Op het DEC System-10 bijvoorbeeld bestaat de identificatie van de gebruiker uit de combinatie (*projectnummer,gebruikersnummer*) en alle gebruikers met hetzelfde projectnummer zijn gedoemd partners te zijn wat betreft de bestandsbenadering. Deze techniek is duidelijk niet goed genoeg als verschillende, maar elkaar overlappende, partnerschappen voor verschillende bestanden nodig zijn.

De beschermingsmethode met behulp van gebruikersklassen werd oorspronkelijk ontworpen in het midden van de jaren '60 voor de Atlas 2, die zeven verschillende toegangsprivileges en vier gebruikersklassen had (een partnerschap werd expliciet aangegeven door

de gebruiker); het wordt nu op diverse systemen toegepast, zoals het DEC System-10, dat acht toegangsprivileges en drie gebruikersklassen heeft (het partnerschap wordt impliciet aangegeven).

Een algemenere techniek die de hiërarchische beperkingen van gebruikersklassen overwint, is die waarbij de eigenaar in zijn GBI alle gebruikers die hij toegangsprivileges heeft gegeven op een lijst aangeeft, samen met een specificatie van die privileges. Als een enkele lijst gekoppeld is aan de gehele GBI, dan zijn de toegewezen privileges noodzakelijkerwijs hetzelfde voor alle bestanden; een alternatieve oplossing is die waarbij een aparte lijst is gekoppeld aan alle ingangen in de GBI, zodat verschillende privileges voor verschillende bestanden gegeven kunnen worden aan verschillende gebruikersgroepen. In het tweede geval kan er veel ruimte door de lijsten worden ingenomen en moet deze methode niet gebruikt worden, tenzij de verhoogde flexibiliteit essentieel is.

Een variatie op deze techniek, die in MULTICS (Daley en Neuman, 1965) gebruikt wordt, is het door de eigenaar geven van toestemming aan andere gebruikers voor het maken van koppelingen in hun GBI's naar ingangen in zijn eigen GBI. Figuur 7.3 laat gebruiker Fred zien, die een koppeling heeft naar een bestandsingang in de GBI van een andere gebruiker, Jan. De koppeling staat Fred toe het bijbehorende bestand, dat van Jan is, te benaderen. De GBI van Jan zal een lijst bevatten van alle gebruikers die hij toestaat dergelijke koppelingen te maken. Een nadeel van deze methode is dat bij het wissen van een bestand ook alle koppelingen gewist moeten worden, wat weer inhoudt dat het systeem een manier moet hebben voor het vinden van alle koppelingen. Als alternatief hiervoor kan een telling bijgehouden worden van het aantal koppelingen en een bestand kan alleen gewist worden als de telling nul is.



Figuur 7.3 *Koppeling tussen GBI's*

Samengevat bieden de hierboven beschreven technieken diverse verschillende oplossingen voor het probleem van gemeenschappelijk gebruik en beveiliging. In het algemeen kan men stellen dat, wanneer de toegepaste methode flexibeler is, het extra gebruik van ruimte groter is en de tijd die voor benadering nodig is langer.

7.4 DE ORGANISATIE VAN HET SECUNDAIRE GEHEUGEN

Zoals in sectie 7.1 werd opgemerkt, wordt de ruimte voor het opslaan van bestanden meestal toegewezen in blokken van een vaste grootte en omdat de grootte van bestanden variabel is, is er voor zowel de bestanden als de vrije ruimte een of andere dynamische opslagtechniek nodig. Er zijn diverse manieren voor het organiseren van de blokken binnen een bestand, waarvan we er drie hieronder beschrijven.

(1) Aaneenschakelen van blokken

Een woord in ieder blok van het bestand wordt gebruikt als een wijzer naar het volgende blok (zie figuur 7.4). De ingang in de GBI voor het bestand wijst naar het eerste blok in de keten. De extra ruimte die nodig is, is één woord per blok voor ieder bestand.

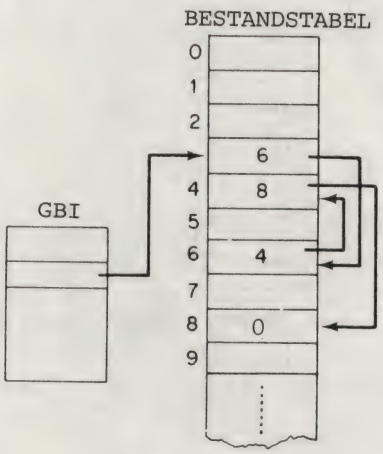
Figuur 7.4 *Aaneengeschakelde bestandsblokken*

Een nadeel van deze methode is het aantal schijfhandelingen dat nodig is voor het vinden van het einde van het bestand. Dat kan bijzonder onhandig zijn als het bestand gewist moet worden en de beslagen ruimte teruggegeven moet worden aan een lijst voor de vrije ruimte; de benodigde veranderingen aan wijzers voor het teruggeven van de ruimte vertrouwen op het feit dat ze de plaats van het einde van het bestand kennen. Daarom wordt de ingang in de GBI vaak zo uitgebreid, dat deze zowel naar het eerste als naar het laatste blok verwijst.

De lezer zal opmerken dat de toegang tot het bestand noodgedwongen sequentieel is, omdat alle blokken alleen bereikt kunnen worden door het naar onderen aflopen van de keten. Deze methode van aaneenschakeling is daarom het best geschikt voor situaties waarin de bestanden sequentieel verwerkt worden. In die gevallen bestaat het extra werk voor het krijgen van toegang tot het bestand alleen uit het inlezen van de opeenvolgende blokken.

(2) Bestandstabel

Voor deze methode van bestandskoppeling wordt de toestand van de schijf vastgelegd in een *bestandstabel*, waarin ieder blok op de schijf door een woord weergegeven wordt. De ingang in de GBI voor een bestand wijst naar de plaats in de bestandstabel die het eerste blok van het bestand weergeeft. Deze plaats wijst op zijn beurt naar de plaats in de tabel die het volgende blok weergeeft, en zo verder (zie figuur 7.5). Het laatste blok in het bestand wordt weergegeven door een loze wijzer. Het bestand dat in figuur 7.5 getoond is, beslaat dus de blokken 3, 6 en 8 van de schijf. De extra ruimte die bij deze methode nodig is, is een woord voor ieder schijfblok.



Figuur 7.5 *Bestandstabel*

Net zoals bij de methode van aaneenschakeling van blokken is de toegang tot het bestand noodgedwongen sequentieel. Als hulp bij het uitbreiden en wissen van een bestand kan de ingang in de GBI een wijzer naar de plaats in de bestandstabel bevatten, die het laatste blok van het bestand weergeeft.

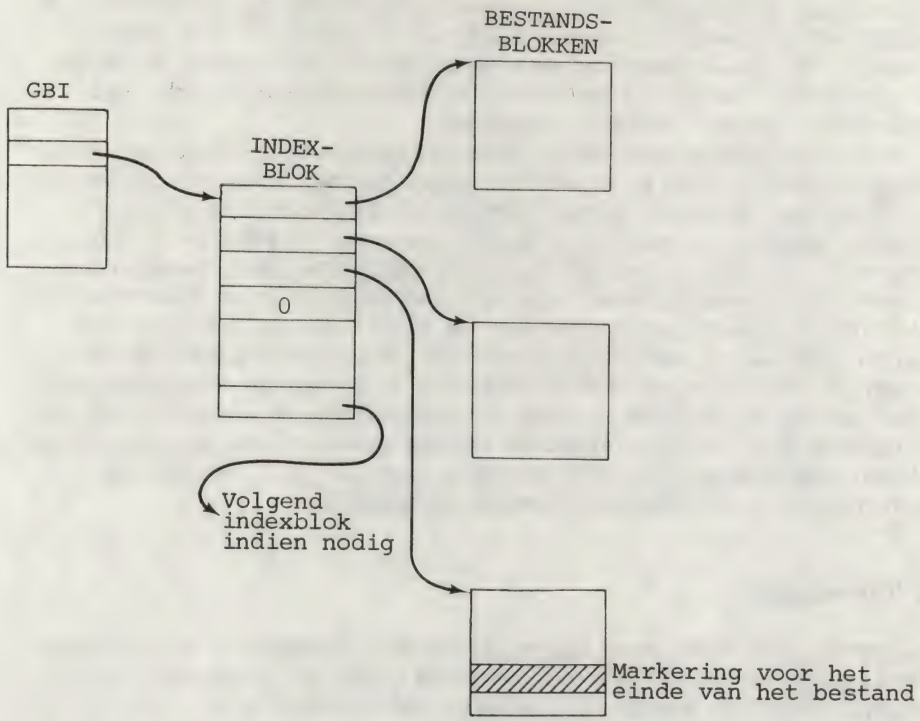
Aangezien de bestandstabel meestal te groot is om in het hoofdgeheugen vastgehouden te worden, moet het op de schijf zelf opgeslagen worden en moet het met één blok tegelijk in het geheugen gebracht worden, al naar gelang dat verlangd wordt. Dit betekent dat er voor het lezen van een bestand van N blokken N extra schijfbenaderingen nodig kunnen zijn voor het lezen van de juiste stukken in de bestandstabel. De extra last zal alleen minder dan dat zijn, wanneer een aantal van de plaatsen die de opeenvolgende blokken van het bestand weergeven in hetzelfde blok van de bestandstabel blijken te liggen. Hierom verdient het duidelijk de voorkeur om de ruimte, die door ieder bestand in beslag genomen wordt, zo aaneengesloten mogelijk te houden, in plaats van toe te staan dat het bestand over de gehele schijf wordt verspreid.

(3) Indexblokken

De koppelingswijzers voor elk bestand zijn opgeslagen in een apart *indexblok* op de schijf. Als het bestand groot is, kunnen er meerdere indexblokken nodig zijn, waarbij de ene steeds aan de volgende gekoppeld is (zie figuur 7.6). De ingang in de GBI voor het bestand wijst naar het eerste indexblok in de keten. Aangezien het laatste indexblok voor een bestand waarschijnlijk niet volledig gebruikt is, is de gemiddelde extra benodigde ruimte iets meer dan één woord per blok voor ieder bestand. De extra benodigde ruimte is bij kleine bestanden naar verhouding groter dan bij grote.

Het grote voordeel van indexblokken is, dat de bestanden niet sequentieel benaderd hoeven te worden; ieder blok kan willekeurig benaderd worden door het simpelweg aangeven van de bestandsnaam en een offset (de verwijzing ten opzichte van het begin van de index) in het indexblok. Deze methode van bestandskoppeling is daarom geschikt voor die situaties waarin de bestanden een interne structuur bezitten en in situaties waarin individuele onderdelen apart benaderd moeten worden.

De extra tijd die nodig is bij het benaderen van het bestand is gering: de eerste toegang vereist dat het indexblok gelezen wordt, maar bij volgende toegangen is er geen extra tijd nodig, tenzij er andere indexblokken nodig zijn. Het toevoegen of verwijderen van blokken, middenin het bestand, houdt echter een herschikking van de wijzers in de indexblokken in. Dit kan een vrij tijdrovende handeling zijn en dat maakt deze opslagmethode ongeschikt voor bestanden die regelmatig veranderen.



Figuur 7.6 *Het gebruik van indexblokken*

Terloops merken we op dat bij elk van de hierboven genoemde methodes het einde van het bestand met een speciale markering, die in het laatste blok geplaatst is, aangegeven kan worden. Dit kan bij ieder soort bestand gedaan worden, waarbij we een unieke markering kunnen definiëren die niet als een gegeven geïnterpreteerd kan worden. Zo zou bijvoorbeeld de markering voor het einde van het bestand, bij bestanden die uit leestekens bestaan, een bitpatroon kunnen zijn dat geen van de bekende leestekens weergeeft. Een dergelijke markering kan echter niet voor een binair bestand gedefinieerd worden, omdat alle bitpatronen geldige gegevens zijn. In dit geval is het nodig dat de lengte van het bestand (in woorden of bytes) opgeslagen wordt als een bijkomend onderdeel van zijn ingang in de GBI. (Dit is waarschijnlijk sowieso een goed idee, omdat het een extra controle kan geven op een eventuele verminking van een bestand.)

Als laatste opmerking over de koppeling van bestandsblokken melden we nog dat een systeem meerdere koppelingsvormen kan toestaan, in het bijzonder als een deel van de bestanden waarschijnlijk sequentieel en een ander deel niet sequentieel benaderd zal gaan worden. De benodigde vorm van de koppeling kan door de gebruiker aangegeven worden bij de aanmaak van het bestand, terwijl door het

systeem in een geschikte, van tevoren ingestelde vorm voorzien wordt.

Een even belangrijk aspect van de organisatie van het secundaire geheugen is het beheer van de ongebruikte ruimte. Een methode is het beschouwen van de vrije blokken als één bestand dat zij zelf hebben gedefinieerd, en ze aan elkaar te koppelen met een van de hierboven beschreven technieken. Als de techniek met de indexblokken gebruikt wordt, moeten alle handelingen (dat wil zeggen het toewijzen en terugsturen van vrije blokken) die op een vrije keten uitgevoerd worden, plaatsvinden aan het einde van die keten; want als ze aan het begin zouden plaatsvinden, zou dat de herschikking inhouden van een groot aantal wijzers. Als een techniek met aaneenschakeling of met bestandstabellen gebruikt wordt, kunnen de handelingen aan elk uiteinde van de keten plaatsvinden. De techniek met indexblokken heeft als verder nadeel dat er, als er *N* blokken toegewezen of teruggestuurd worden, ook *N* wijzers uit het laatste indexblok verwijderd of in het laatste indexblok toegevoegd moeten worden. Ter vergelijking: er hoeven maar twee wijzers veranderd te worden als een van de andere twee technieken toegepast wordt, onafhankelijk van het aantal blokken dat toegewezen of teruggestuurd wordt. (Als er bijvoorbeeld blokken teruggestuurd worden naar het einde van de vrije keten, zijn er twee wijzers die veranderd moeten worden: een uit het laatste blok van de oude keten naar het eerste blok van de toevoeging en een van de systeemindex naar het einde van de nieuwe keten.)

Een andere methode voor het vastleggen van de vrije ruimte is het gebruik van een *bittabel*, dit is een gedeelte van het geheugen waarin elk bit een blok op de schijf weergeeft. Als het blok vrij is, wordt het bijbehorende bit op nul gezet; als het blok gebruikt wordt, wordt het bit op één gezet. Voor het vinden van *N* vrije blokken is het slechts nodig dat de bittabel afgezocht wordt naar de eerste *N* bits die op nul staan en dat er daarna een eenvoudige berekening uitgevoerd wordt voor het verkrijgen van de bijbehorende blokadressen.

In bepaalde situaties kan de bittabel zo groot zijn dat het onhandig is deze nog in het hoofdgeheugen te houden. In die gevallen kan de tabel opgeslagen worden in het secundaire geheugen, terwijl maar een sectie ervan in het hoofdgeheugen aanwezig is. Deze sectie kan voor alle toewijzingen gebruikt worden en wanneer hij vol is (dat wil zeggen als alle bits één zijn) kan hij omgewisseld worden met een andere sectie. De teruggave van vrije ruimte betekent dat die sectie van de tabel, die correspondeert met de teruggegeven blokken, teruggehaald moet worden, en dat de juiste bits op nul gezet moeten worden. Dit kan druk verkeer naar de schrijftabel tot gevolg hebben, maar dat kan verminderd worden door het bijhouden van een lijst van alle teruggegeven blokken die dan gebruikt wordt voor het bijwerken van de tabel, telkens wanneer er van die tabel een nieuwe sectie in het hoofdgeheugen komt. Deze techniek moet met zorg toegepast worden zodat, in het geval van een schijfbeschadiging, de inhoud van de bittabel in overeenstemming blijft met de toegewezen ruimte.

De keuze betreffende de grootte van het blok wordt meestal door de ontwerper van de hardware gemaakt (een uitzondering die het vermelden waard is, is de IBM 370 serie, waarbij de gebruiker een aparte keuze voor ieder bestand kan maken). De keuze wordt aan de hand van de volgende criteria gemaakt:

(1) De ruimteverspilling ten gevolge van het feit dat een aantal blokken niet gevuld worden. Dit zal toenemen als de blokgrootte toeneemt.
(2) De verspilling van ruimte ten gevolge van wijzers voor de ketens. Hoe kleiner de blokken zijn, des te meer wijzers er nodig zijn.
(3) De eenheden waarin het opslagapparaat gegevens overbrengt naar het hoofdgeheugen. De blokgrootte moet een veelvoud zijn van deze eenheid om ervoor te zorgen dat iedere gegevensoverdracht volledig benut wordt.
(4) De hoeveelheid geheugen die nodig is voor iedere schijf- of leeshandeling op een bestand. Afhankelijk van de methode van de organisatie van de schijf, kan het nodig zijn dat er ruimte gevonden moet worden in het geheugen voor het huidige indexblok of het huidige blok van de bestandstabel, en het blok dat beschreven of gelezen wordt.

Veel voorkomende blokgroottes liggen tussen de 512 en 2048 bytes.

Strategieën voor het toewijzen van ruimte kunnen variëren tussen rechttoe-rechtaan (gebruik het eerste vrije blok van de keten vrije blokken, of het eerste blok in de bittabel) en zeer verfijnd (gebruik het blok dat het aantal bewegingen van de lees/schrijfkop bij het lezen zo laag mogelijk houdt). De extra benodigde tijd voor het lezen van bestanden waarvan de blokken over de hele schijf verspreid zijn, kan vrij aanzienlijk zijn en kan leiden tot verzadiging van het schijfkanaal. Hij kan teruggebracht worden door de hierboven geschetste strategie of door het periodiek langsgaan van de bestandsstructuur, waarbij de bestanden zoveel mogelijk verdicht worden. Deze tweede techniek kan ingebouwd worden in de back-up procedures (het maken van kopieën), die in de volgende sectie beschreven worden.

7.5 DE ONSCHENDBAARHEID VAN OPSLAGSYSTEMEN

Omdat de inhoud van een bestand het werk van maanden kan weergeven of onvervangbare gegevens kan bevatten, is het essentieel dat een opslagsysteem voor bestanden voorziet in afdoende mechanismen voor het maken van reserve-kopieën en het terughalen en herstellen van bestanden, voor het niet onwaarschijnlijke geval dat er een storing in de hard- of software optreedt. Dit betekent dat het systeem duplicaat-kopieën moet onderhouden van alle bestanden, zodat zij hersteld kunnen worden na een onfortuinlijke gebeurtenis.

Er zijn twee principieel verschillende manieren voor het maken van kopieën. De eerste van de twee staat bekend als de *periodieke* (of *algehele*) *dump* (opslag). Met een bepaalde tijdsinterval wordt de inhoud van alle bestanden 'gedumpt' (opgeslagen) op het een of andere medium; meestal is dat een magnetische band. In het eventuele geval van een storing kunnen alle bestanden gereconstrueerd worden tot de staat waarin ze waren op het moment dat de laatste dump gemaakt werd. Losse bestanden die onbedoeld gewist zijn, kunnen teruggehaald worden door middel van het aflopen van de gehele dump-band. De nadelen van een periodieke opslag zijn:

(1) Het kan nodig zijn het opslagsysteem gedurende het maken van de dump buiten werking te stellen. Het alternatief is het niet dumpen van bestanden die openstaan, zodat ze beschreven mogen worden.
(2) Het maken van de dump neemt meestal veel tijd in beslag (20 minuten tot 2 uur, afhankelijk van de grootte van het systeem en de snelheid van de bandopname-apparatuur). Dit betekent dat het dumpen niet te vaak plaats kan vinden en daarom kan het gebeuren dat een bestand dat teruggehaald is niet meer actueel is.

De tweede en verfijndere techniek is de *gedeeltelijke dump* (Fraser, 1969). Bij deze techniek wordt alleen een dump gemaakt van die informatie die veranderd is nadat de vorige dump gemaakt werd. Dit betekent dat alleen bestanden die aangemaakt of gewijzigd zijn en ingangen in indexen die veranderd zijn bij iedere gelegenheid gedumpt worden. De omvang van dergelijke informatie zal gering zijn en daardoor kunnen de dumps frequent gemaakt worden. Om te bepalen welke bestanden gedumpt moeten worden, wordt een vlag gezet in de ingang in de index van ieder bestand, telkens wanneer het bestand gewijzigd wordt. Deze wordt natuurlijk weer teruggezet als het bestand gedumpt wordt. Om ervoor te zorgen dat het dumpproces niet alle ingangen in de indexen hoeft te controleren, kan er een algemene vlag gezet worden in iedere GBI om aan te geven of er individuele vlaggen gezet zijn in de GBI. Aangezien het dumpproces alle bestanden overslaat waarvan de vlag niet gezet is - de bestanden die op het ogenblik bijgewerkt worden vallen daar ook onder - kan het parallel lopen met het normale werk. Het is zelfs mogelijk, zoals in MULTICS, dat het dumpproces een lage prioriteit heeft, altijd in de machine aanwezig is en voortdurend de indexen afzoekt naar bijgewerkte informatie.

De nadelen van de gedeeltelijke dump liggen in de grote hoeveelheden gegevens die worden aangemaakt en in de complexiteit van de herstelprocedure. Na een storing moet de informatie gereconstrueerd worden uit de opeenvolging van gebeurtenissen die op de band zijn opgeslagen. De banden zijn in tegengestelde chronologische volgorde samengesteld en het opslagsysteem wordt herleid naar de situatie zoals die was op het moment dat de laatste periodieke dump gemaakt werd. (Periodieke dumps zullen zelden gemaakt worden, een normale

frequentie is één keer per week.) Gedurende deze procedure zoekt het proces, dat het herladen verzorgt, alleen die bestanden van de dumpbanden op die het niet al eerder teruggeladen heeft, zodat recente bestanden niet overschreven worden door oudere versies. Ten slotte wordt de laatste band met een periodieke dump gebruikt voor het completeren van de herstel-operatie. Deze methode is beter dan het starten met de band met de periodieke dump en het voorwaarts werken, omdat er geen overbodige informatie teruggeladen wordt.

Merk op dat, wat de dumpmethode ook is, het herstellen van het systeem een goede gelegenheid is voor het verdichten van bestanden tot aaneengeschakelde blokken. Dit vermindert extra last bij het benaderen van bestanden, maar gaat ten koste van het enigszins verlengen van de herstelprocedure.

7.6 HET OPENEN EN SLUITEN VAN BESTANDEN

We merkten in het vorige hoofdstuk op dat, als een I/O stroom gekoppeld is aan een bestand, het besturingssysteem dan het apparaat en de plaats moet opzoeken van het opgeslagen bestand. Deze procedure staat bekend als het *openen* van het bestand; de omgekeerde procedure, het *sluiten* van het bestand, wordt expliciet uitgevoerd als alle I/O naar het bestand voltooid is, of impliciet uitgevoerd als het proces dat het bestand gebruikt afloopt.

Voor het implementeren van deze handelingen in het besturingssysteem hebben we twee procedures nodig die als volgt gevormd zijn:

open (bestandsnaam, manier) *sluit (bestandsnaam)*

Hierin is *bestandsnaam* de naam van het te openen of te sluiten bestand en *manier* is het soort toegang dat gevraagd wordt (bijvoorbeeld lezen, schrijven of aanmaken). De handelingen die door *open* worden verricht zijn:

(1) Zoek de ingang voor het bestand in de index op.
(2) Controleer of het verzoekende proces veroorzaakt werd door een gebruiker met de bijbehorende privileges voor toegang tot de opgegeven 'manier'.
(3) Voer controles uit om ervoor te zorgen dat, als het bestand al open is voor lezen (mogelijk door een ander proces), het nu niet geopend kan worden voor schrijven. En zorg ervoor dat, als het al open is voor schrijven, het nu helemaal niet geopend kan worden. Deze controles zullen straks besproken worden.
(4) Stel het apparaat en de plaats vast waar het bestand opgeslagen is. Als het bestand nieuw aangemaakt moet worden, dan wordt de plaats aangegeven door de routine daarvoor.

(5) Maak een *bestandsbeschrijver*, die alle relevante informatie bevat over het bestand, voor toekomstige gegevensoverdrachten. De bestandsbeschrijver wordt, zoals in sectie 6.5 aangegeven is, door I/O procedure gebruikt als een bron van informatie voor het uitvoeren van I/O verzoeken, en vermijdt op die manier de noodzaak voor het uitvoeren van de open-procedure voor elke gegevensoverdracht die op het bestand betrekking heeft.

De informatie die nodig is in de bestandsbeschrijver omvat:

(1) de bestandsnaam;
(2) het adres van de apparaatbeschrijver van het apparaat waarop het bestand is opgeslagen;
(3) de plaats van het eerste blok in het bestand;
(4) de plaats van het volgende blok dat gelezen of geschreven moet worden (aangenomen dat de toegang sequentieel is);
(5) de manier van toegang.

Er wordt naar de bestandsbeschrijver verwezen vanuit de juiste stroombeschrijver van het proces dat het bestand heeft geopend, zoals in figuur 6.4a is weergegeven.

De lees/schrijf uitsluiting waarnaar in deel (3) van de beschrijving van *open* werd verwezen, kan geïmplementeerd worden door het opnemen van twee extra onderdelen in de ingang in de index van ieder bestand. De eerste is een 'schrijfbit' dat altijd gezet wordt wanneer het bestand voor schrijven geopend wordt; de tweede is een 'gebruiksteller' voor het aantal processen dat op het ogenblik het bestand voor lezen open heeft. Een proces mag het bestand alleen voor lezen openen als het schrijfbit nul is en mag het alleen voor schrijven openen als het schrijfbit en de gebruiksteller beide nul zijn. Helaas ligt er een probleem in deze techniek door de mogelijkheid dat meerdere processen tegelijkertijd het bestand willen openen of sluiten, en daardoor tegelijkertijd veranderingen maken in de ingang in de index. Dit kan voorkomen worden door het schrijven van de *open* en *sluit* procedures als kritische sectoren, die ieder omsloten zijn door dezelfde seinpalen voor wederzijdse uitsluiting. Dat heeft echter het ernstige nadeel dat, als een proces onderbroken wordt tijdens het openen of sluiten van een bestand, deze handelingen dan ontoegankelijk worden voor andere processen.

Een alternatieve techniek is het maken van een lijst van alle bestandsbeschrijvers van alle bestanden, die op een bepaald apparaat open zijn. Er kan naar deze lijst verwezen worden vanuit de apparaatbeschrijver van het betrokken apparaat. Als er een bestand geopend moet worden, inspecteert de *open*procedure de lijst op het eventueel al aanwezig zijn van bestandsbeschrijvers voor het bestand, en als dat zo is, inspecteert deze de toegangsmanier die zij aangeven. De procedure weigert het bestand te openen, tenzij er voldaan wordt aan de lees/schrijf beperkingen. Een seinpaal voor wederzijdse uitsluiting kan toegepast worden voor de bescherming van de lijst tegen samenvallende benaderingen van processen die hetzelfde bestand op

dezelfde tijd willen openen of sluiten. Het nadeel van deze techniek ligt in de tijd die er nodig is voor de inspectie van de lijst, die in een groot systeem met meervoudige toegang gemakkelijk 100 beschrijvers zou kunnen bevatten. Deze tijd is in het bijzonder kritisch omdat de lijst maar door één proces tegelijkertijd kan worden geïnspecteerd, want indien het proces onderbroken wordt, is de lijst ontoegankelijk voor andere processen.

Een gedeeltelijke oplossing voor deze problemen is het onderverdelen van elke bestandsbeschrijver in twee gescheiden structuren: een *centrale bestandsbeschrijver* voor ieder open bestand en een *plaatselijke bestandsbeschrijver* voor ieder proces dat gebruik maakt van het bestand. Een plaatselijke bestandsbeschrijver wordt steeds aangemaakt wanneer een proces een bestand opent en alle plaatselijke bestandsbeschrijvers voor een bepaald bestand wijzen naar de ene centrale bestandsbeschrijver voor dat bestand. De centrale bestandsbeschrijvers zijn gekoppeld aan de juiste apparaatbeschrijver (zie figuur 7.7). Een centrale bestandsbeschrijver bevat informatie die hetzelfde is voor ieder proces dat het bestand gebruikt, namelijk:

(1) de bestandsnaam;
(2) het adres van de apparaatbeschrijver van het apparaat waarop het bestand is opgeslagen;
(3) de plaats van het eerste blok in het bestand;
(4) de gebruiksteller voor het bestand;
(5) het schrijfbit voor het bestand.

Een plaatselijke bestandsbeschrijver bevat informatie die speciaal bestemd is voor het proces dat het bestand gebruikt, namelijk:

(1) de plaats van het volgende blok dat gelezen of geschreven moet worden;
(2) de toegangsmanier,

plus een wijzer naar de centrale bestandsbeschrijver voor het bestand. Het invoeren van centrale bestandsbeschrijvers elimineert de opslag van dubbele informatie in aparte beschrijvers en vermindert de lengte van de lijst met beschrijvers voor ieder apparaat. Het vermindert ook de hoeveelheid werk die de *open*- en *sluit*procedures moeten verzetten bij het afzoeken van de lijst met beschrijvingen. Het is van groter belang dat het de toepassing van wederzijdse uitsluiting mogelijk maakt op elke beschrijver apart in plaats van op de hele lijst als geheel; het voorkomt zodoende het gevaar voor het ontstaan van de eerder genoemde knelpunten. De seinpaal voor de wederzijdse uitsluiting kan opgeslagen worden in de centrale beschrijver voor ieder bestand.

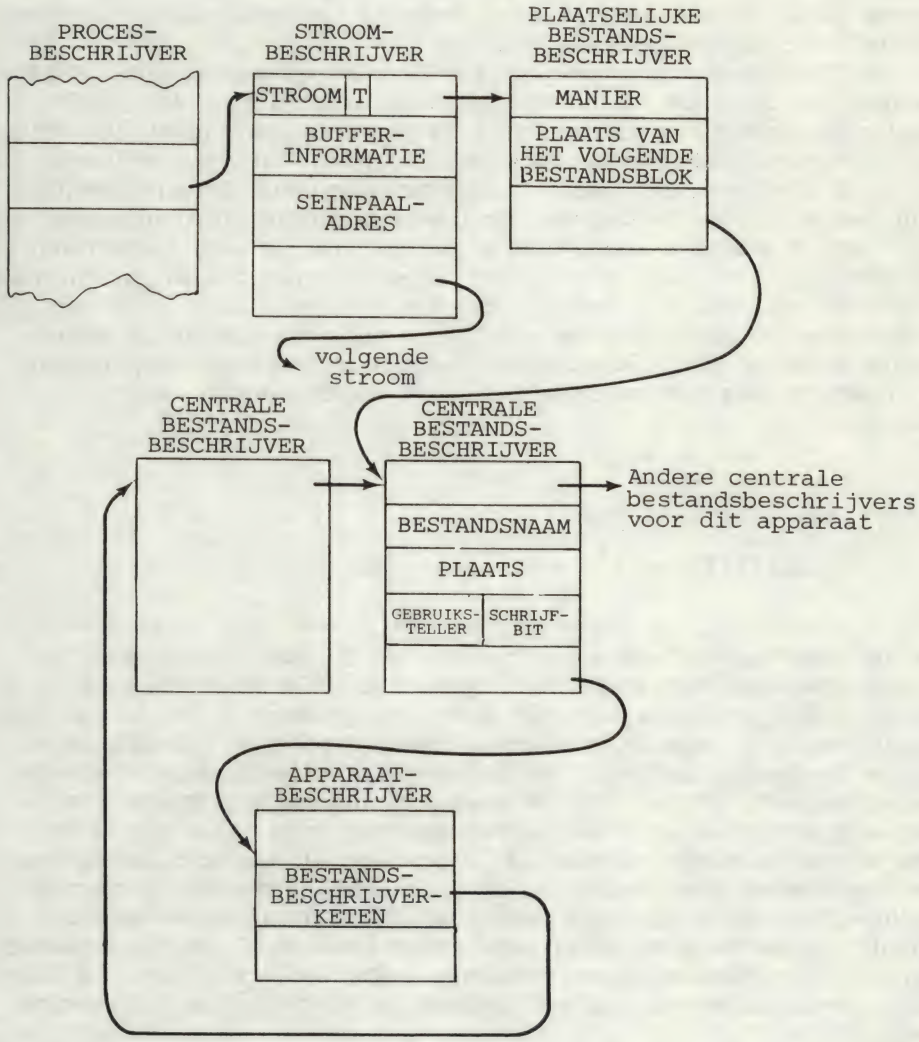
Een variatie op deze techniek (Courtois e.a., 1971) is het vervangen van het schrijfbit in de centrale beschrijver van het bestand door nog een seinpaal *w*. De *open*- en *sluit*procedures zijn dan zo geschreven dat zij de volgende stukken code bevatten.

open

```
als manier = lezen dan
begin passeer(blokkeer);
      gebruikstel. := gebruikstel. + 1
      als gebruikstel. = 1 dan passeer(w);
      verhoog(blokkeer)
einde
anders passeer(w)
```

sluit

```
als manier = lezen dan
begin passeer(blokkeer);
      gebruikstel. = gebruikstel. + 1
      als gebruikstel. = 0 dan verhoog(w);
      verhoog(blokkeer)
einde
anders passeer(w)
```



Figuur 7.7 *Plaatselijke en centrale bestandsbeschrijvers*

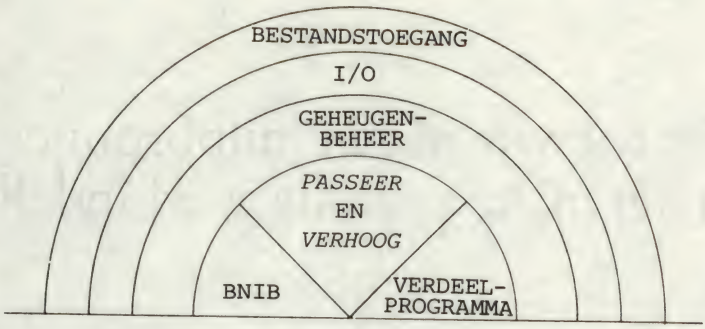
Bij deze variatie worden de toegangsverzoeken die geweigerd worden als logisch gevolg hiervan in een wachtrij voor de seinpaal (*w*) geplaatst, terwijl in de oorspronkelijke versie de systeemontwerper de vrijheid houdt om, afhankelijk van de toepassing, de verzoeken al dan niet in een wachtrij te zetten.

De *sluit*procedure is relatief eenvoudig. Deze bestaat uit het wissen van de plaatselijke bestandsbeschrijver en uit het verlagen van de gebruikersteller in de centrale bestandsbeschrijver. Als de gebruiksteller nul is wordt de centrale bestandsbeschrijver ook gewist en wordt de index, indien nodig, bijgewerkt (als bijvoorbeeld het bestand net aangemaakt is).

Het wissen van een bestand kan beschouwd worden als het aanroepen van *open* met de juiste parameter voor *manier*. Dit heeft echter het nadeel dat het verzoek om te wissen geweigerd zal worden als het bestand al open is, wat vaak het geval zal zijn met bestanden uit de bibliotheek. Dit maakt de vervanging van bibliotheekbestanden wat moeizaam. Een manier om dit probleem te omzeilen is het opnemen in iedere centrale beschrijvingstabel van een 'wisverzoek aangevraagd bit', dat gezet wordt als een wisverzoek is gedaan voor een open bestand. De *sluit*procedure controleert dit bit: als het gezet is en de gebruiksteller gaat naar nul, dan kan het bestand gewist worden. Het wissen wordt bespoedigd als *open* toegang weigert tot een bestand waarvan het 'wisverzoek aangevraagd bit' gezet is.

7.7 TEN SLOTTE

In dit hoofdstuk hebben we verschillende aspecten besproken van opslagsystemen voor algemene toepassingen. De volgende stap in de samenstelling van ons papieren besturingssysteem is het in gebruik nemen van een bepaald systeem dat gebaseerd is op de technieken die we besproken hebben. Dat zal ons een volgende laag voor onze 'ui' opleveren. Die bestaat uit code voor het implementeren van bepaalde veiligheids- en integriteitsmechanismen, voor het bijhouden van vrije ruimte en voor het openen en sluiten van bestanden. De permanente gegevensstructuren die erbij komen zijn bestandsindexen en een structuur voor het vastleggen van de toegewezen ruimte. Tijdelijke structuren, die alleen bestaan als er een bestand open is, zijn plaatselijke en centrale bestandsbeschrijvers. De huidige stand van zaken van het systeem wordt in figuur 7.8 weergegeven.



Figuur 7.8 *De huidige staat van het papieren besturingssysteem*

8 Het toewijzen van hulpbronnen en het maken van de werkindeling

Tot aan dit punt in de ontwikkeling van ons papieren besturingssysteem hebben we aangenomen dat alle processen de hulpbronnen en faciliteiten bezitten die ze nodig hebben. Het wordt nu tijd om te gaan bekijken hoe processen hun hulpbronnen verkrijgen en hoe een beperkte hoeveelheid hulpbronnen zo goed mogelijk verdeeld kan worden tussen de verschillende processen. Onze bespreking van de toewijzing van hulpbronnen zal ook het maken van de werkindeling (Engels: scheduling) omvatten, omdat deze twee functies nauw met elkaar verbonden zijn: besluiten over de prioriteit van processen kunnen afhangen van de verplichtingen die hulpbronnen hebben en de invoering van nieuwe processen in de machine wordt duidelijk beïnvloed door de hoeveelheid vrije capaciteit van de hulpbronnen. Het maken van een werkindeling kan, omdat deze betrokken is bij de toewijzing van de centrale processoren, in feite beschouwd worden als een deel van de bespreking van de toewijzing van hulpbronnen in het algemeen.

8.1 ALGEMENE WAARNEMINGEN

In een situatie waar hulpbronnen onbeperkt aanwezig zijn, zou een 'pak het als je het nodig hebt' verwervingsmethode volledig acceptabel zijn. Helaas is het zelden lonend om in hulpbronnen te voorzien voor het voldoen aan alle gelijktijdige verzoeken van alle processen in het systeem. Er moeten dus technieken uitgedacht worden voor het verdelen van een beperkte hoeveelheid hulpbronnen tussen een aantal processen die erom strijden. De doelstellingen van deze technieken zijn:

(1) het wederzijds uitsluiten van processen bij ondeelbare hulpbronnen;
(2) het voorkomen van het vastlopen van het systeem (zie sectie 3.2) ten gevolge van verzoeken om toewijzing van hulpbronnen;
(3) het zorgen voor een hoge mate van gebruik van de hulpbronnen;

(4) het geven van de gelegenheid aan alle processen voor het, binnen een redelijke termijn, verkrijgen van toegang tot de hulpbronnen.

De lezer zal merken dat deze doelstellingen niet altijd met elkaar verenigbaar zijn. In het bijzonder kan het tevreden stellen van de gebruiker, aangegeven door doelstelling (4), meestal slechts bewerkstelligd worden door te beknibbelen op doelstelling (3). Dit komt omdat de gemiddelde wachttijd, voordat er voldaan wordt aan een verzoek voor de toewijzing van een hulpbron, langer wordt bij een hogere bezettingsgraad van die hulpbron. Het vinden van een evenwicht tussen het tevreden stellen van de gebruiker en het gebruik van de hulpbronnen is een van de criteria aan de hand waarvan de ontwikkeling voor het toewijzen van hulpbronnen en het beleid voor een werkindeling kan plaatsvinden. In een direct verwerkend systeem met een gegarandeerde responstijd kan men verwachten dat de balans doorslaat naar de kant van de gebruiker; in een batchsysteem zou deze ten gunste van een hoge gebruiksgraad kunnen doorslaan. Dit kan voor het beheer grote problemen veroorzaken bij een systeem dat tracht te voorzien in service voor zowel batch- als meervoudige toegang.

Het is nuttig de toewijzing van hulpbronnen onder twee koppen te bespreken: de *mechanismen* en de *beleidsvormen*. Met *mechanismen* bedoelen we het 'bouten-en-moeren' aspect van de manier waarop de toewijzing gemaakt wordt. Dit omvat zaken als gegevensstructuren voor het beschrijven van de toestand van de hulpbronnen, technieken die voor het exclusieve gebruik van ondeelbare hulpbronnen zorgen, en middelen voor het in wachtrijen zetten van verzoeken voor het gebruik van hulpbronnen die niet onmiddellijk gehonoreerd kunnen worden. De *beleidsvormen* besturen de manier waarop de mechanismen toegepast worden. Zij zijn betrokken bij de beslissing of honoreren van verzoeken verstandig is, zelfs als de juiste hulpbronnen beschikbaar zijn. Dit aspect omvat ook vragen betreffende het vastlopen van het systeem en het evenwicht in het systeem: een onverstandige beslissing kan een situatie tot gevolg hebben waarin een aantal processen niet verder kan of waarin een systeem overbelast is met betrekking tot een klasse hulpbronnen. Op dit laatste punt ontstaat de koppeling tussen het toewijzen van hulpbronnen en het maken van de werkindeling, aangezien de kans op overbelasting verminderd kan worden door weloverwogen besluiten aangaande de prioriteit van processen en het invoeren van nieuwe processen in het systeem. We zullen in de volgende secties zowel de mechanismen als de werkindeling bespreken.

8.2 MECHANISMEN VOOR DE TOEWIJZING

Uit wat we hiervoor gezien hebben moet duidelijk zijn dat elk element van een computersysteem, dat beperkt verkrijgbaar is en dat gedeeld moet worden, als een hulpbron beschouwd kan worden. In feite vallen hulpbronnen in een van de ondergenoemde categorieën. We zullen de toewijzingsmechanismen voor iedere categorie bespreken waarbij rekening gehouden wordt met: welk deel van het systeem de toewijzing maakt, de gegevensstructuren die gebruikt worden voor het beschrijven van de hulpbronnen, en de manier waarop processen in een wachtrij gezet worden als zij op toewijzing wachten.

(1) Centrale processoren

We hebben in hoofdstuk 4 al besproken hoe een processor door het verdeelprogramma toegewezen wordt aan het eerste niet lopende proces in de processorwachtrij. De gegevens die een processor beschrijven kunnen in een processorbeschrijver staan die vergelijkbaar is met de apparaatbeschrijver zoals gebruikt wordt voor de gegevens die betrekking hebben op een randapparaat (zie hoofdstuk 6). De processorbeschrijver zou normaliter het volgende kunnen bevatten:

(a) de identificatie van de processor;
(b) de huidige toestand - of die nu de gebruikerstoestand of de supervisortoestand aangeeft;
(c) een wijzer naar de procesbeschrijver van het huidige proces.

De processorbeschrijvers kunnen in de centrale tabel gehouden worden of er kan vanuit die tabel naar verwezen worden.

Bij een configuratie waarin de processoren onderling verschillen, kunnen de processorbeschrijvers uitgebreid worden zodat zij een identificatie bevatten betreffende de individuele eigenschappen van de processor (bijvoorbeeld of deze hardware heeft voor de weergave van getallen met drijvende komma). In dit geval kan het zijn dat iedere processor het best geschikt is voor het uitvoeren van een bepaald soort processen en daarom zijn eigen processorwachtrij kan hebben waarnaar verwezen wordt vanuit zijn beschrijver. De processorwachtrijen zijn dan vergelijkbaar met de verzoekwachtrijen voor randapparatuur. De toewijzing van processen aan verschillende processorwachtrijen is een onderwerp waarvan grote delen nog niet onderzocht zijn en we zullen er hier niet verder op ingaan.

(2) Geheugen

In hoofdstuk 5 zagen we hoe geheugen door het geheugenbeheer van het besturingssysteem toegewezen kan worden aan systemen

zowel met als zonder pagina-indeling, De gegevensstructuren die de toestand van het geheugen beschrijven zijn de paginatabellen, de segmenttabellen of de lijsten van de vrije blokken, afhankelijk van de betrokken systeemarchitectuur. Een proces dat wacht op de overdracht van een nieuwe pagina of segment uit het secundaire geheugen, wordt onuitvoerbaar gemaakt, terwijl de statusbits in zijn procesbeschrijver de reden aangeven. Verzoeken voor een pagina of een segment wordt door het systeem voor het geheugenbeheer in wachtrijen opgesteld als I/O verzoekblokken en worden afgehandeld door de apparaatbesturing voor het juiste apparaat voor het secundaire geheugen.

(3) Randapparatuur

We beschreven in hoofdstuk 6 hoe in systemen met op stromen gebaseerde I/O een ondeelbaar apparaat aan een proces wordt toegewezen wanneer de betrokken stroom geopend wordt. Bij systemen die geen stromen gebruiken kan een randapparaat na een rechtstreeks verzoek aan het besturingssysteem toegewezen worden. In beide gevallen is het toewijzingsmechanisme hetzelfde. De gegevensstructuur die een randapparaat beschrijft, is zijn apparaatbeschrijver en de processen die op toewijzing van een apparaat wachten kunnen in een wachtrij voor een seinpaal opgesteld worden die in de beschrijver opgenomen is. Wederzijdse uitsluiting voor het gebruik van een randapparaat wordt verzekerd door het geven van de beginwaarde 1 aan de seinpaal.

(4) Achtergrondgeheugen

Achtergrondgeheugen, dat gebruikt wordt bij de implementatie van een virtueel geheugen, wordt toegewezen door het beheersysteem voor het geheugen. Het achtergrondgeheugen dat als bestandsruimte gebruikt wordt, wordt toegewezen door het opslagsysteem. Sommige besturingssystemen, in het bijzonder MULTICS, maken geen onderscheid tussen bestanden en segmenten, zodat de opslag van de bestanden een deel wordt van het virtuele geheugen en zodat het opslagsysteem verantwoordelijk is voor alle toewijzingen. De gegevensstructuur die gebruikt wordt voor de toewijzing, is een soort lijst van vrije blokken of een bittabel zoals beschreven in sectie 7.4.

Verzoeken voor bestandsruimte worden alleen geweigerd als een individuele gebruiker zijn toegewezen deel overschreden heeft, of als het hele achtergrondgeheugen vol is. In geen van de twee gevallen is het verstandig om de verzoeken in een wachtrij op te stellen; de eerste situatie kan alleen door ingrijpen van de gebruiker zelf opgelost worden, bij de tweede situatie kan niet gegarandeerd worden dat deze binnen een korte tijd zal veranderen.

(5) Bestanden

Een individueel bestand kan beschouwd worden als een hulpbron, in die zin dat meerdere processen er mogelijk gemeenschappelijk gebruik van willen maken. Zolang alle processen zodanig werken dat er alleen gelezen wordt, is de hulpbron deelbaar; als een proces in het bestand wil schrijven, is de hulpbron ondeelbaar.

De toewijzing van een bestand aan een proces wordt verzorgd door het opslagsysteem als het bestand geopend wordt; het uitsluiten van het schrijven kan bereikt worden met behulp van de methoden die in sectie 7.6 beschreven zijn. De gegevensstructuren die de bestanden beschrijven zijn natuurlijk de bestandsindexen.

Batchsystemen zetten bestandsverzoeken meestal niet in een wachtrij omdat gelijktijdige (niet-lees) verzoeken meestal een gebruikersfout aangeven. Bij het verwerken van transacties of bij procesbesturing kan het echter zeer verstandig zijn om binnen het opslagsysteem wachtrijen te maken voor de toegang. Deze wachtrijen kunnen gekoppeld zijn aan seinpalen, zoals in sectie 7.6 is beschreven.

Het is uit het bovenstaande duidelijk dat toewijzingsmechanismen op de diverse niveaus van het besturingssysteem geïmplementeerd zijn, waarbij ieder mechanisme op het niveau aangetroffen wordt dat geschikt is voor de hulpbron die toegewezen wordt. We zullen echter zien dat het beleid voor het gebruik van de mechanismen door het gehele systeem heen hetzelfde moet zijn.

8.3 HET VASTLOPEN VAN HET SYSTEEM

De eerste beleidsvormen voor de toewijzing van hulpbronnen die we gaan bekijken hebben te maken met het probleem van het vastlopen van het systeem (Engels: deadlock), zoals dat in sectie 3.2 beschreven werd.

Als hulpbronnen alleen op basis van hun beschikbaarheid aan processen worden toegewezen, kan het vastlopen gemakkelijk plaatsvinden. In zijn eenvoudigste vorm zal het vastlopen in de volgende omstandigheden voorkomen: proces *A* heeft hulpbron *X* toegewezen gekregen en vraagt om hulpbron *Y*; proces *B* heeft hulpbron *Y* toegewezen gekregen en vraagt om hulpbron *X*. Als beide hulpbronnen ondeelbaar zijn en als geen van de processen de hulpbron vrijgeeft die het bezet, dan is vastlopen van het systeem het gevolg. In het algemeen zijn de voorwaarden die nodig en voldoende zijn voor het vastlopen (Coffman e.a., 1971) de volgende:

(1) de betrokken hulpbronnen moeten ondeelbaar zijn;

(2) de processen houden de hulpbronnen die zij al toegewezen hebben gekregen vast, terwijl ze op nieuwe wachten;
(3) hulpbronnen kunnen niet voortijdig weggehaald worden als ze in gebruik zijn;
(4) er bestaat een cirkelvormige keten van processen waarbij elk proces de hulpbronnen vasthoudt waarnaar het volgende proces in de keten op dat moment vraagt.

Het probleem van het vastlopen kan opgelost worden door het gebruik van een van de volgende strategieën:

(1) voorkom vastlopen door er te allen tijde voor te zorgen dat er aan minsten één van de vier hierboven genoemde voorwaarden niet voldaan wordt;
(2) herken het vastlopen als het voorkomt en probeer het dan te herstellen;
(3) vermijd het vastlopen door een passende anticiperende handeling.

We bekijken deze strategieën één voor één.

(1) Het voorkómen van het vastlopen

Voor het voorkómen van het vastlopen moet het onmogelijk gemaakt worden om aan ten minste één van de hierboven genoemde voorwaarden te voldoen.

Het voldoen aan voorwaarde (1) is moeilijk te voorkomen omdat sommige hulpbronnen (bijvoorbeeld een kaartlezer of een bestand waarin geschreven kan worden) door hun aard ondeelbaar zijn. Het gebruik van spooling (zie sectie 6.6) kan het gevaar voor vastlopen bij ondeelbare randapparatuur wegnemen.

Het voldoen aan voorwaarde (2) kan onmogelijk gemaakt worden door te stellen dat processen al hun hulpbronnen in één keer moeten aanvragen, en door te stellen dat zij niet verder kunnen voordat aan alle verzoeken voldaan is. Dit heeft het nadeel dat hulpbronnen, die maar een korte tijd gebruikt worden, toch toegewezen worden en daardoor voor langere tijd ontoegankelijk zijn. Het is echter gemakkelijk te implementeren en kan op den duur goedkoper blijken te zijn dan een complexer algoritme.

Het voldoen aan voorwaarde (3) is gemakkelijk onmogelijk te maken door het stellen van de regel dat, indien een proces een verzoek om toegang tot een hulpbron geweigerd wordt, het dan alle hulpbronnen moet vrijmaken die het op dat moment in gebruik heeft en ze, als dat nodig is, opnieuw moet aanvragen samen met de extra hulpbronnen. Helaas kan deze draconische strategie in de praktijk nogal onhandig zijn, omdat het voortijdig loskoppelen van (bijvoorbeeld) een regeldrukker het door elkaar mengen van de uitvoer van verschillende opdrachten tot gevolg zou hebben. Bovendien zou, zelfs als de hulpbron op een behoorlijke manier los te koppelen is,

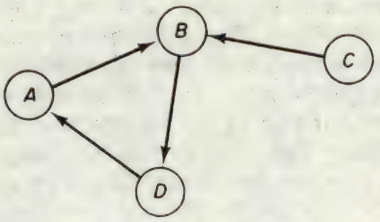
het extra werk, veroorzaakt door het opslaan en weer herstellen van de toestand van de hulpbron ten tijde van het loskoppelen, vrij aanzienlijk kunnen zijn. In het geval van een centrale processor is het extra werk echter vrij gering (namelijk het opslaan van de vluchtige omgeving van het huidige proces) en de processor kan altijd beschouwd worden als ontkoppelbaar.

Het voldoen aan voorwaarde (4) kan onmogelijk gemaakt worden door het geven van een zodanige volgorde aan de soorten hulpbronnen dat, als er aan aan proces hulpbronnen van het soort k zijn toegewezen, het alleen nog maar mag verzoeken om die hulpbronnen die na k komen in de volgorde. Dit zorgt ervoor dat de situatie, waarin er in een cirkel op elkaar gewacht wordt, nooit kan voorkomen. Het nadeel is de druk die geplaatst wordt op de natuurlijke volgorde van de verzoeken om hulpbronnen, alhoewel die verlicht kan worden door veelgebruikte hulpbronnen aan het begin van de volgorde te plaatsen.

Een interessant voorbeeld van deze drie manieren voor het voorkómen van het vastlopen van het systeem doet zich voor in OS/360 (Havender, 1968). In dit systeem worden taken onderverdeeld in 'taakstappen' en er is een starter voor de taakstappen die de verantwoordelijkheid heeft voor het verkrijgen van de hulpbronnen die in elke taakstap nodig zijn. Het gelijktijdig laten beginnen van verschillende taakstappen voor verschillende taken wordt bereikt door het hebben van meerdere kopieën van de starter voor de taakstappen. Het vastlopen van de kopieën onderling wordt voorkomen door ervoor te zorgen dat elke starter de hulpbronnen altijd verwerft in de volgorde: bestanden, geheugen, randapparatuur. Het elkaar blokkeren van verschillende taakstappen, die dezelfde bestanden willen benaderen, wordt voorkomen door ervoor te zorgen dat de starter de bestanden, die hij voor de gehele taak nodig heeft, in één keer verwerft. Er moet opgemerkt worden dat het voortijdig loskoppelen in dit geval niet gepast zou zijn, omdat een taak niet graag zou zien dat er met zijn bestanden geknoeid wordt door vreemde taakstappen, op het moment dat hijzelf tussen taakstappen in is. Ook kan er geen redelijke volgorde gemaakt worden in de verzoeken voor toegang tot bestanden. Het vastlopen door randapparatuur wordt voorkomen door taken te dwingen hun hulpbronnen (uitgezonderd bestanden) los te laten tussen taakstappen en ze weer te laten aanvragen bij de volgende stap, indien ze nodig zijn.

(2) Het herkennen en herstellen van vastlopen

Als de beleidsvormen voor het voorkómen van het vastlopen als te beperkend worden beschouwd, kan een andere benadering acceptabel zijn. Deze geeft de mogelijkheid tot vastlopen, maar vertrouwt erop dat dit herkend wordt en hersteld kan worden. De waarde van deze benadering hangt af van de frequentie van het vastlopen en van het soort herstel dat mogelijk is.



Figuur 8.1 *Voorbeeld van een toestandstekening (hulpbronnen A, B en D zijn betrokken bij het vastlopen)*

Algoritmen voor het herkennen van het vastlopen werken op de herkenning van het wachten in een cirkel, zoals is uitgedrukt in voorwaarde (4) hierboven. De toestand van het systeem kan op elk moment weergegeven worden door een *toestandstekening*, waarin de knooppunten hulpbronnen zijn en waarin een pijl tussen knooppunten *A* en *B* betekent dat er een proces bestaat dat hulpbron *A* bezet en vraagt om hulpbron *B* (zie figuur 8.1).

De cirkelvormige wachttoestand wordt weergegeven door een gesloten lus in de toestandstekening (bijvoorbeeld *A*, *B*, *D* in figuur 8.1). Het algoritme voor de herkenning onderhoudt een afbeelding van de toestandstekening in een geschikte vorm en controleert deze op het eventuele bestaan van gesloten lussen. De controle kan plaatsvinden na iedere toewijzing of, omdat het extra werk dat hierbij betrokken is waarschijnlijk teveel is, na een vast tijdsinterval.

Het herkennen van het vastlopen van het systeem is alleen zinnig als er een redelijke poging tot herstel gedaan kan worden. De definiëring van 'redelijk' kan, afhankelijk van de situatie, uitgebreid worden zodat deze de technieken erbij insluit die hieronder in volgorde van verfijning opgesomd worden.

(a) Stoot alle vastgelopen processen af. Dit komt grofweg overeen met het postkantoor dat zo nu en dan een zak post verbrandt als het overbelast wordt, maar het is de methode die toegepast wordt in de meeste systemen voor algemeen gebruik.
(b) Herstart de vastgelopen processen vanuit een tussenliggend controlepunt, als dat bestaat. Als dit op een te simplistische manier wordt toegepast, kan het rechtstreeks terugleiden naar het oorspronkelijke punt waar het vastlopen optrad, maar de flexibele respons van het systeem zorgt ervoor dat dit meestal niet gebeurt.
(c) Stoot de vastgelopen processen één voor één af totdat het systeem niet langer meer vast staat. De volgorde van het afstoten kan zodanig zijn dat de investeringsverliezen van hulpbronnen die al gebruikt zijn zo gering mogelijk zijn. Deze benadering houdt in dat na elke afstoting het algoritme voor de herkenning weer aangeroepen moet worden, om te bekijken of het systeem nog steeds vast staat.

(d) Koppel de hulpbronnen los van de vastgelopen processen totdat zij niet langer vast staan. Evenals in (c) kan de volgorde van het loskoppelen zo zijn dat de verlieskosten van de een of andere functie zo laag mogelijk zijn. Het weer aanroepen van het algoritme voor de herkenning is na iedere loskoppeling nodig. Een proces dat voortijdig van een hulpbron losgekoppeld wordt, moet later weer verzoeken om de toewijzing van die hulpbron.

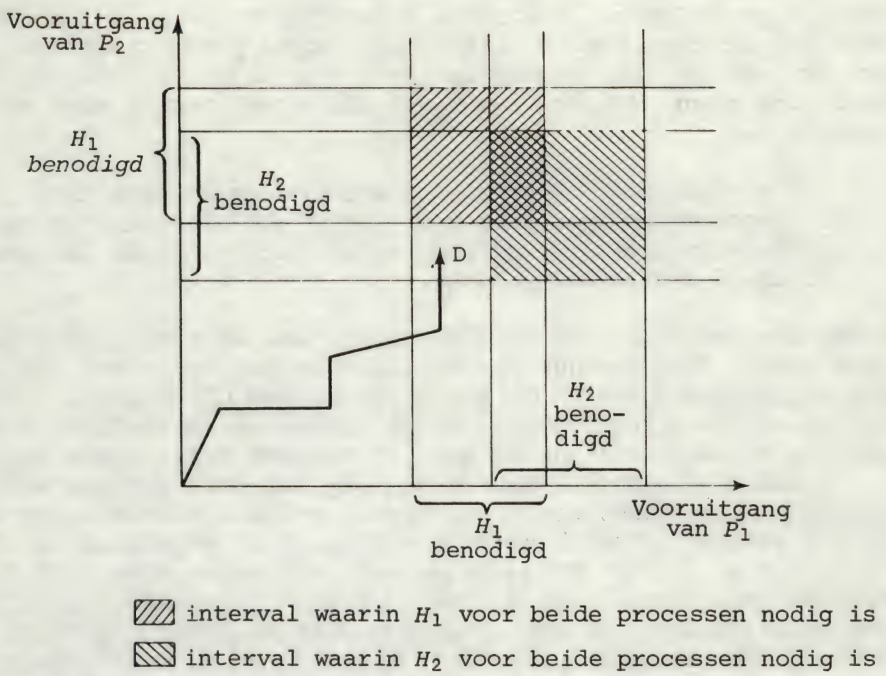
Voordat we het onderwerp verlaten, moeten we opmerken dat de herkennning van het vastlopen vaak overgelaten wordt aan de computerbeheerders in plaats van dat het door het systeem zelf wordt uitgevoerd. Een attente beheerder zal uiteindelijk opmerken dat bepaalde processen lijken te blijven hangen en bij nader onderzoek zal hij zich realiseren dat het systeem is vastgelopen. De gebruikelijke herstelhandeling is het stoppen en herstarten (indien mogelijk) van de vastgelopen processen.

(3) Het vermijden van vastlopen

Met het vermijden van vastlopen bedoelen we het gebruik van een algoritme dat vooraf opmerkt dat het vastlopen waarschijnlijk gaat optreden en dat daarom een verzoek om toewijzing van een hulpbron zal weigeren dat anders gehonoreerd zou zijn. Dit is een duidelijk onderscheid van het voorkómen van het vastlopen, wat er a priori voor zorgt dat het niet kan gebeuren door het onmogelijk maken van het voldoen aan een van de noodzakelijke voorwaarden.

Een verleidelijke benaderingswijze voor het ontwerpen van algoritmen voor het vermijden van vastlopen is als volgt. Verander vóórdat een verzoek toegewezen wordt, bij wijze van proef, de toestandstekening naar de situatie zoals die zou zijn als het verzoek zou zijn toegewezen, en pas dan het algoritme voor de herkenning van vastlopen toe. Als het herkenningsalgoritme zijn toestemming geeft, honoreer het verzoek dan; weiger het verzoek als dat niet het geval is en herstel de toestandstekening in zijn oude vorm. Helaas zal deze techniek niet altijd werken omdat het van de stelling uitgaat dat, als een toewijzing in het vastlopen uitmondt, dat onmiddellijk zal gebeuren. Dat deze stelling niet opgaat, kan duidelijk gemaakt worden door het bestuderen van figuur 8.2 (Dijkstra), die een systeem met twee processen en twee hulpbronnen weergeeft.

De gezamenlijke vooruitgang van de processen P_1 en P_2 wordt weergegeven door het tekenen van een traject, dit is in het diagram de dikke lijn. De horizontale delen van het traject geven de periodes weer dat P_1 loopt, de verticale delen geven de periodes weer dat P_2 loopt en de gearceerde delen geven de periodes weer dat P_1 en P_2 tegelijk lopen. (In een configuratie met slechts één processor kunnen er alleen horizontale en verticale delen bestaan.) Het wordt het traject verboden het gearceerde deel op het diagram in te gaan, omdat tijdens dat interval ten minste één van de hulpbronnen H_1 en H_2 voor beide processen nodig is. Het belangrijkste punt is dat,



Figuur 8.2 *Illustratie van het vastlopen*

indien het traject gebied D ingaat, vastlopen onvermijdelijk is, omdat er dan geen mogelijkheid meer bestaat dat het traject kan voorkómen dat het bij het gearceerde deel komt (het traject kan alleen naar boven of naar rechts bewegen in de richting van een positieve vooruitgang). Het vastlopen komt echter in gebied D zelf niet voor en het binnengaan van D zou door geen enkel herkenningsalgoritme gevonden worden. Zodoende zou het gebruik van een herkenningsalgoritme in dit geval het vastlopen niet voorkomen. Om het anders te stellen: een herkenningsmechanisme kan maar één stap vooruit kijken, terwijl enkele stappen verder het vastlopen onvermijdelijk kan zijn.

Uit de bovenstaande bespreking blijkt duidelijk dat een goed werkend vermijdingsalgoritme enige voorkennis moet bezitten over het mogelijke patroon van toekomstige gebeurtenissen, zodat het het mogelijk vastlopen kan opmerken voordat dat werkelijk optreedt. Het soort voorkennis dat verondersteld werd is een hulpmiddel dat het ene algoritme van het andere onderscheidt. We bespreken hieronder het *bankiersalgoritme*, dat waarschijnlijk het bekendste voorbeeld is.

De voorkennis die voor het bankiersalgoritme nodig is, is de maximale hoeveelheid gebruik die een proces tijdens zijn bestaan van een hulpbron gaat maken. We noemen deze hoeveelheid de *claim*

van een proces op een hulpbron. In batchsystemen is deze hoeveelheid vrij acceptabel omdat taakomschrijvingen gebruikt kunnen worden voor het van tevoren aangeven van de maximum aanspraak op hulpbronnen. Het algoritme geeft alleen toestemming voor een verzoek als:

(a) het verzoek plus het huidige gebruik onder de claim ligt, en
(b) er hierna een volgorde bestaat waarop alle processen tot een einde kunnen komen, zelfs als zij allemaal aanspraak kunnen maken op hun volledige claim.

Zodoende controleert een toewijzingsalgoritme vóór een toewijzing of er genoeg hulpbronnen overblijven zodat het maken van een dergelijke volgorde mogelijk is. Het is aangetoond (Habermann, 1969) dat het werk dat betrokken is bij het maken van de controle, evenredig is met de wortel uit het aantal processen in het systeem.

Bekijk als voorbeeld voor de werking van het algoritme nog eens figuur 8.2. De claims van P_1 en P_2 hebben beide betrekking op H_1 en H_2. Voordat gebied D wordt ingegaan, is H_1 toegewezen aan P_1 en is H_2 vrij. Er zijn zodoende genoeg hulpbronnen over voor het voltooien van zowel P_1 als P_2, zelfs als beide processen hun volledige claim opeisen. De juiste volgorde is in dit geval P_1 gevolgd door P_2. Binnenin het gebied D is H_1 aan P_1 en H_2 aan P_2 toegewezen. Als dus beide processen hun volledige claim opeisen, is er geen volgorde (noch P_1 voor P_2 noch omgekeerd) waarin zij voltooid kunnen worden. Het bankiersalgoritme weigert dus de toegang tot gebied D door het niet honoreren van het verzoek van P_2 voor H_2. P_2 zal tijdelijk opgeschort worden en alleen P_1 zal kunnen lopen. Het traject zal daarom een horizontale lijn volgen langs de onderkant van het gearceerde gebied, totdat P_1 H_2 loslaat. Op dit punt kan het oorspronkelijke verzoek van P_2 voor H_2 gehonoreerd worden omdat er nu voldoende hulpbronnen beschikbaar zijn voor het voltooien van beide processen.

Er moet opgemerkt worden dat het bankiersalgoritme altijd uitgaat van het ergste geval (dat waarin alle processen hun volledige claim opeisen). Het is daardoor mogelijk dat er soms een verzoek afgewezen wordt omdat vastlopen eventueel *zou* kunnen optreden, terwijl het vastlopen in werkelijkheid niet voorgekomen zou zijn als het verzoek was toegewezen. De toegang tot gebied D in figuur 8.2 wordt bijvoorbeeld verboden vanwege de mogelijkheid dat P_1 zijn claim op H_2 en P_2 zijn claim op H_1 zouden kunnen doen gelden. Als in werkelijkheid een van de twee processen het verkiest zijn claim niet te doen gelden op een erg ongeschikt moment, dan zou gebied D totaal veilig zijn (omdat de gearceerde gebieden niet de aangegeven plaatsen zouden innemen), en dan zou de weigering van het verzoek van P_2 voor H_2 een onnodige beperking geweest zijn. De neiging van het bankiersalgoritme tot overdreven voorzichtigheid, samen met het extra werk dat ervoor het toepassen van het algoritme nodig is, zijn de belangrijkste beperkingen bij het gebruik ervan.

We sluiten onze bespreking van het vastlopen af met het bekijken op welk niveau in het besturingssysteem beleidsvormen voor het voorkomen ervan geïmplementeerd moet worden.

Als een strategie voor het voorkómen gebruikt wordt, moeten alle delen van het systeem die hulpbronnen toewijzen onder één of meer van de eerder beschreven beperkingen werken. De beperkingen kunnen op de juiste niveaus als deel van de toewijzingsmechanismen geïmplementeerd worden.

Als herkenning gebruikt wordt, zijn er geen wijzigingen van de toewijzingsmechanismen vereist. Het herkenningsalgoritme, dat toegang moet hebben tot alle gegevensstructuren die de toewijzing verzorgen, kan op een hoog niveau geïmplementeerd worden, mogelijk in de werkindeler zelf. (De werkindeler wordt in volgende secties beschreven.)

In het geval van het vermijden van vastlopen moeten alle verzoeken voor toegang tot een hulpbron nauwkeurig onderzocht worden door het vermijdingsalgoritme. We zullen in de volgende sectie zien dat alle verzoeken voor hulpbronnen via de werkindeler gedaan kunnen worden en deze werkindeler wordt dan de natuurlijke plaats voor de implementatie van het algoritme.

8.4 DE WERKINDELER

Onder het indelen van werk verstaat men gewoonlijk het vraagstuk betreffende het invoeren van een nieuw proces in het systeem en de volgorde waarin processen uitgevoerd zouden moeten worden. Zoals eerder opgemerkt werd, zijn deze zaken nauw verbonden met de toewijzing van hulpbronnen. Het is in de praktijk zo dat beslissingen over het maken van de werkindeling en over het toewijzen van hulpbronnen zo sterk met elkaar betrokken zijn, dat het vaak verstandig is de verantwoordelijkheid voor beide in handen van een systeemproces te leggen. Dit proces zullen we de *werkindeler* (Engels: *scheduler*) noemen. De functies van de werkindeler zijn hieronder beschreven.

(1) Het invoeren van nieuwe processen

In een batchsysteem worden de taken die op uitvoering wachten, opgeslagen in een *takenpot* die in het secundaire geheugen gehouden wordt. (Hoe ze daar komen wordt in hoofdstuk 11 besproken.) De werkindeler begint de uitvoering van een taak met het starten van het juiste proces, zoals het vertalen. De keuze wat de volgende taak is die uitgevoerd moet worden, hangt af van de hulpbronnen die ieder proces nodig heeft (zoals vermeld in zijn taakbeschrijver) en van het huidige patroon van de toewijzing van de hulpbronnen in

het systeem. Om een hoge verwerkingssnelheid te krijgen, moet de werkindeler een nieuw proces starten zodra de capaciteit van de hulpbronnen dat toelaat.

In een meervoudig toegankelijk systeem worden processen aangemaakt als gebruikers zich inschrijven (Engels: log in) in het systeem. Iedere nieuwe gebruiker verhoogt het aantal aanvragen voor hulpbronnen en toegang kan geweigerd worden als het aantal ingeschreven gebruikers zo groot is dat het de responstijd verhoogt tot de grens van het toelaatbare.

(2) Het geven van prioriteiten aan processen

De volgorde waarin de processen uitgevoerd worden hangt af van òf de volgorde van de processorwachtrij, òf de volgorde waarin het verdeelprogramma ze selecteert uit de wachtrij. We suggereerden in hoofdstuk 4 dat voor het zo laag mogelijk houden van het extra werk dat betrokken is bij het toewijzen van processoren aan processen, het verdeelprogramma altijd het eerste geschikte proces uit de wachtrij moet nemen; dit impliceert dat de volgorde in de rij de bepalende factor zal zijn. De werkindeler is verantwoordelijk voor het geven van prioriteiten aan de processen, zodat ze op de juiste plaats in de wachtrij gezet worden wanneer ze uitvoerbaar worden. Algoritmen voor het bepalen van geschikte prioriteiten zullen in de volgende sectie besproken worden.

(3) Implementatie van beleidsvormen voor hulpbron-toewijzing

De hier bedoelde beleidsvormen zijn betrokken bij het vermijden van het vastlopen van het systeem en bij de balancering van het systeem; dat wil zeggen dat zij moeten zorgen voor het voorkómen van over- of onderbelasting van de diverse soorten hulpbronnen. Criteria voor het bepalen van de balans zullen in de volgende sectie besproken worden.

Omdat het gedrag van het systeem voornamelijk bepaald wordt door de activiteiten van de werkindeler, is het belangrijk dat de werkindeler een hoge prioriteit heeft ten opzichte van andere processen. Als de werkindeler de hoogste prioriteit heeft, zal het verdeelprogramma hieraan, iedere keer dat hij uitvoerbaar is, de voorkeur geven boven andere processen. Dit verzekert dat het systeem vlot reageert op veranderde omstandigheden, in het bijzonder op veranderde eisen. De gelegenheden waarbij de werkindeler geactiveerd zou kunnen worden, zijn:

(1) er wordt een hulpbron gevraagd;
(2) er komt een hulpbron vrij;
(3) een proces loopt af;

(4) er komt een nieuwe taak in de takenpot (of een nieuwe gebruiker probeert zich in te schrijven).

Het werkt verduidelijkend om de analogie op te merken tussen interrupts en de bovenstaande gebeurtenissen die te maken hebben met de werkindeler. Beide komen op onvoorspelbare tijdstippen voor en beide kunnen ervoor zorgen dat het systeem zijn gedrag wijzigt. Bij gebeurtenissen rondom de werkindeling vinden de wijzigingen op een hoog niveau plaats; wijzigingen ten gevolge van interrupts vinden op een laag niveau plaats en hebben gevolgen voor de uitvoerbaarheid van processen en voor de toewijzing van centrale processoren. Van interrupts kan verwacht worden dat zij om de paar milliseconden voorkomen, terwijl gebeurtenissen bij de werkindeling waarschijnlijk om de paar seconden voorkomen.

Als de werkindeler met zijn werk klaar is zet hij zichzelf tijdelijk buiten werking door het uitvoeren van een *passeer*handeling op een seinpaal; deze seinpaal krijgt een verhoogsignaal als er een gebeurtenis plaatsvindt die gevolgen heeft voor de werkindeler. In een besturingssysteem dat het vastlopen noch voorkomt noch vermijdt, is het mogelijk dat geen gebeurtenis plaatsvindt met gevolgen voor de werkindeler. In deze situatie is het van vitaal belang dat de werkindeler weer tot leven geroepen wordt om vast te stellen dat vastlopen heeft plaatsgevonden; dit kan gedaan worden door het uitvoeren van een *verhoog*handeling op de seinpaal door een klokinterrupt, bijvoorbeeld elke tien seconden.

Het aanvragen en vrijgeven van hulpbronnen (de eerste twee van de bovenstaande gebeurtenissen met betrekking tot werkindeling) kunnen geïmplementeerd worden door het besturingssysteem te voorzien van de volgende twee procedures:

verzoek hulpbron (hulpbron,resultaat)
ontkoppel hulpbron (hulpbron)

Deze procedures plaatsen de noodzakelijke informatie over de betrokken hulpbron en over het aanroepende proces in een gegevensgebied dat voor de werkindeler toegankelijk is en geven dan een verhoogsignaal aan de seinpaal van de werkindeler. De werkindeler zal reageren op een manier die past bij de criteria die we in de volgende sectie zullen bespreken. De tweede parameter van *verzoek hulpbron* wordt gebruikt voor het overbrengen van het resultaat van het verzoek en kan ook een aanwijzing geven omtrent de reden van een eventuele weigering. Merk op dat een verzoek, dat in eerste instantie geweigerd is door de werkindeler, in een wachtrij gezet kan worden totdat de hulpbron beschikbaar komt. In dit geval zal het verzoek uiteindelijk gehonoreerd worden en de *resultaat*-parameter van *verzoek hulpbron* zal dat aangeven. Het in een wachtrij zetten en de bijbehorende vertraging zijn duidelijk voor het verzoekende proces.

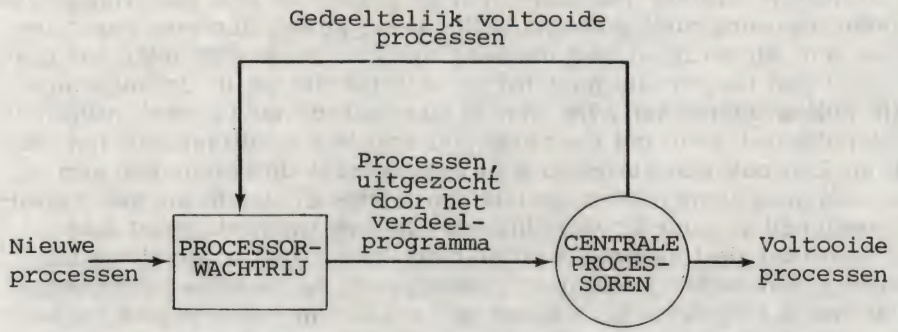
Een nadeel van het combineren van de werkindeler samen met de functies voor de toewijzing van hulpbronnen in één enkel proces

is de extra last die door het overschakelen tussen processen veroorzaakt wordt, wat steeds nodig is als een hulpbron toegewezen of ontkoppeld wordt. Als men de extra last zwaarder vindt wegen dan de winst die verkregen kan worden uit de systeembalans en het goede gebruik van hulpbronnen, dan kunnen de twee functies gescheiden worden. In dit geval voeren de routines *verzoek hulpbron* en *ontkoppel hulpbron* de functies voor het toewijzen van hulpbronnen uit zonder dat zij de werkindeler daarin betrekken.

8.5 ALGORITMEN VOOR WERKINDELINGEN

Het algemene doel van een algoritme voor een werkindeler is het zo indelen van het werkpatroon van het computersysteem, dat een bepaalde meetwaarde voor de tevredenstelling van de gebruikers zo hoog mogelijk wordt. Deze meetwaarde kan van systeem tot systeem verschillen: in een situatie met batchverwerking kan het de totale verwerkingshoeveelheid zijn, of de gemiddelde verwerkingstijd die voor een bepaalde taakklasse nodig is; in een systeem met meervoudige toegang kan het de gemiddelde responstijd zijn die de gebruiker geboden wordt, of de responstijd die geboden wordt voor bepaalde soorten interactie. Welk algoritme voor de werkindeler gebruikt wordt hangt natuurlijk van het gestelde doel af.

We beginnen met het bekijken van de algoritmen die gebaseerd zijn op de aanname dat de centrale processoren de belangrijkste hulpbronnen zijn. We hebben aan het begin van dit hoofdstuk al aangegeven dat deze aanname in vele gevallen niet juist is en dat andere hulpbronnen ook belangrijk kunnen zijn. Door het bestuderen van processor-gerichte systemen kunnen we echter meer inzicht krijgen in ingewikkelder gevallen die later besproken zullen worden.



Figuur 8.3 *Werkindelingsmodel van een processor-gericht systeem*

Een algemeen model van een processor-gericht systeem is weergegeven in figuur 8.3. Nieuwe processen komen binnen bij de processorwachtrij en worden afgehandeld door een aantal processoren. (We nemen aan dat de processoren identiek zijn.) Na het ontvangen van een bepaalde hoeveelheid verwerkingstijd kan een proces voltooid zijn, in dat geval verlaat dat het systeem; anders wordt het in de wachtrij teruggeplaatst in afwachting van verdere verwerking op een later tijdstip. Het model geeft alleen processen weer die uitvoerbaar zijn of uitgevoerd worden; het weglaten van de geblokkeerde toestand uit hoofdstuk 4 geeft het feit weer dat in processorgerichte systemen alleen maar significante vertragingen kunnen optreden als gevolg van het wachten in de processorwachtrij. Een bepaald algoritme voor de werkindeler wordt gekarakteriseerd door de volgorde van de processen in de wachtrij en de voorwaarden waaronder processen ernaar terug worden geleid. Hieronder bekijken we diverse populaire algoritmen.

(1) Kortste taak eerst

Zoals de naam al zegt is de wachtrij geordend in overeenstemming met de vereiste looptijd voor ieder proces. Het algoritme is alleen geschikt voor batchsystemen, omdat daarin uit de taakomschrijving een schatting verkregen kan worden van de looptijd. Het doel is het minimaliseren van de verwerkingstijd voor korte taken en daarom kan dit het beste gebruikt worden in een situatie waarin de korte taken het gros uitmaken van het grote geheel. In zijn eenvoudigste vorm staat het alle processen toe te lopen tot de voltooid zijn. Er kan echter een wijziging gemaakt worden zodat een nieuw proces een processor voortijdig mag loskoppelen, als dat nieuwe proces een verwerkingstijd heeft die korter is dan die welke nodig is voor het afmaken van het huidige proces. Deze versie, die bekend staat als *pre-emptive shortest job first (voortijdig loskoppelbaar, kortste taak eerst)* bevoordeelt de korte taken nog meer. Een andere wijziging is het geleidelijk verhogen van de prioriteit van een proces, overeenkomend met de tijd die het in de wachtrij heeft gestaan.

(2) De rondlopende wachtrij (Engels: Round Robin)

Dit algoritme werd ontworpen als een middel voor het geven van een snelle respons op korte verzoeken op verwerking, als de looptijden niet vooraf bekend zijn. Van ieder proces wordt een vaste hoeveelheid werk verricht voordat het aan het einde van de wachtrij wordt teruggezet. De wachtrij is daarom cirkelvormig en geordend volgens de tijd die verstreken is sinds er de laatste keer aan gewerkt werd. Processen die een lange verwerkingstijd nodig hebben zullen de wachtrij meerdere malen doorlopen voor zij voltooid worden, terwijl processen waarvan de benodigde tijd korter is dan het quantum, al tijdens de eerste cyclus voltooid zullen worden. Het algoritme voor

de rondlopende wachtrij werd in een van de eerste systemen met tijdsindeling (Engels: time sharing), CTSS (Corbato e.a., 1962), gebruikt en is nadien met verschillende veranderingen in diverse andere systemen geïmplementeerd. De meest veranderingen hebben tot doel dat het instorten van het systeem onder zware last voorkomen wordt. Analyses en praktijk wijzen uit dat, indien de last te zwaar wordt voor een bepaalde quantumgrootte, de prestatie plotseling sterk vermindert. Een manier om dat te voorkomen is het vergroten van het quantum als het aantal processen toeneemt. Dit verhoogt de waarschijnlijkheid dat een proces binnen een quantum voltooid wordt en vermindert daardoor het extra schakelen tussen de processen.

(3) De wachtrij met twee niveaus

Dit is nog een variatie op het algoritme voor de rondlopende wachtrij, die een oplossing tracht te vinden voor het plotseling verminderen van de prestaties onder te zware last. Processen die niet binnen een vaststaand aantal quantums voltooid zijn, worden afgevoerd naar een achtergrondwachtrij, die alleen aandacht krijgt als er geen andere processen in het systeem aanwezig zijn. De achtergrondwachtrij kan zelf behandeld worden op basis van de rondlopende wachtrij, of gewoon afgewerkt worden volgens het principe 'wie het eerst komt, het eerst maalt'. Een systeem met twee wachtrijen wordt vaak gebruikt in situaties waar de batchverwerking en de meervoudige toegang naast elkaar plaatsvinden. De langere batchprocessen hebben dan de neiging naar de achtergrond te verdwijnen en krijgen alleen aandacht als er geen processen aanwezig zijn die vanuit een invoerstation gestart zijn. Als dat gepast lijkt, kan het algoritme natuurlijk uitgewerkt worden tot een *meer-niveau systeem*. Een voorbeeld hiervan zien we in het toezichthoudende programma van het DEC System-10, waar er drie rondlopende wachtrijen zijn met quantums van respectievelijk 0,02 seconden, 0,25 seconden en 2,00 seconden. (De waarden mogen, indien gewenst, zelfs veranderd worden.)

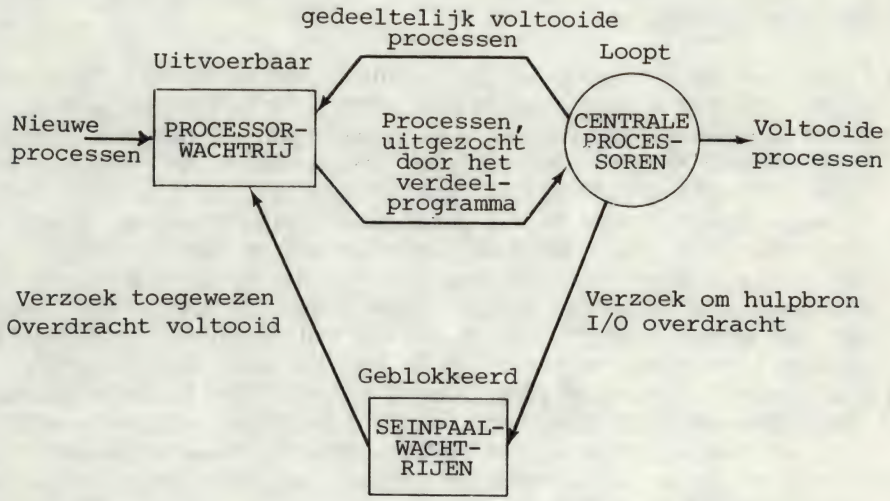
De optimale parameterwaarden, zoals de quantumgrootte en het moment waarop een proces naar de achtergrond verschoven moet worden, zijn voor het rondlopende-wachtrij-algoritme en zijn afstammelingen moeilijk te bepalen (maar bekijk Coffman en Denning, 1973, voor pogingen daartoe). Benaderingen kunnen gevonden worden met behulp van simulatie en kunnen aan de hand van ervaringen over hun werking in de praktijk bijgesteld worden.

De bovenstaande algoritmen vormen de basis van de meest gebruikte beleidsvormen voor het maken van een werkindeling. Andere algoritmen en variaties kan men vinden in het uitstekende overzicht van Coffman en Kleinrock (1968). Een opmerkelijke variatie is die waarbij het van buitenaf opgeven van prioriteiten is toegestaan. Het

gevolg is dat processen die van buitenaf een hoge prioriteit hebben ontvangen, een betere afhandeling krijgen dan het geval zou zijn geweest indien het algoritme puur zou zijn gebruikt. Dit maakt het de gebruikers, die een tijdslimiet of een VIP status hebben, mogelijk een betere respons van het systeem te krijgen, hoewel dit privilege hen in rekening gebracht kan worden.

Systemen waarin de centrale processoren niet de enige hulpbronnen van belang zijn kunnen beschreven worden door het model dat in figuur 8.4 is weergegeven. Dit model verschilt van dat uit figuur 8.3 door het erin betrekken van de geblokkeerde toestand waarin processen wachten op reacties op verzoeken om hulpbronnen, of op I/O opdrachten. Het toevoegen van deze toestand vergroot de complexiteit van het model aanzienlijk. Ten eerste wordt deze toestand niet gekenmerkt door één enkele wachtrij, maar door een veelheid van wachtrijen die betrokken zijn bij de diverse seinpalen die de oorzaak van de blokkade zijn. Ten tweede verhuizen processen van de geblokkeerde naar de uitvoerbare toestand, na gebeurtenissen als het geven van een verhoogsignaal van een seinpaal die op niet te voorspellen momenten plaatsvinden. Het verschuiven van de processen tussen de wachtrijen kan voor de verschillende werkindelingsstrategieën bekeken worden door het maken van òf een formele analyse òf door simulatie (Coffman en Denning, 1973; Svobodova, 1976), alhoewel de complexiteit van het model beide technieken moeilijk toepasbaar maakt.

Zoals de lezer uit figuur 8.4 zal opmaken zou het minimaliseren van het aantal keren dat een proces van de lopende naar de geblokkeerde toestand overgebracht moet worden één van de doelen van



Figuur 8.4 *Algemeen model voor de werkindeling*

het algoritme voor de werkindeling kunnen zijn. Aangezien het overbrengen ten gevolge van I/O overdracht buiten zijn macht ligt, volgt daaruit dat het algoritme het aantal verzoeken om hulpbronnen dat geweigerd wordt zou moeten minimaliseren. Dit betekent dat processen alleen in het systeem ingevoerd zouden moeten worden als er aan hun waarschijnlijke aanspraak op hulpbronnen kan worden voldaan door de hulpbronnen die op het moment beschikbaar zijn. Dit beleid is gemakkelijker te implementeren in batchsystemen, waarin de aanspraak op hulpbronnen van tevoren kan worden aangegeven, dan in meervoudig toegankelijke systemen, waarin de aanspraak schier onvoorspelbaar is. Een voorbeeld van implementatie op een batchsysteem was het besturingssysteem bij Atlas, waarin de taken in drie klassen werden onderverdeeld: korte taken, die weinig procestijd en geen esoterische randapparatuur nodig hebben; taken voor de magnetische band, wat voor zichzelf spreekt; en lange taken, waaronder de rest valt. Hierbij probeert de werkindeler altijd ten minste één taak in ieder van de drie klassen te handhaven.

Opmerkelijk is dat dit soort beleidsvormen niet noodzakelijkerwijs zal leiden tot een hoge bezettingsgraad van de hulpbronnen. De reden daarvan ligt in het feit dat een proces waarschijnlijk niet gedurende zijn gehele bestaan alle hulpbronnen zal gebruiken die het heeft aangegeven. De geclaimde en door de werkindeler toegewezen hulpbronnen kunnen er daarom lange tijd ongebruikt bijliggen. Als een hogere bezetting vereist is, moet de werkindeler niet alleen beleidsbeslissingen nemen over het toewijzen van hulpbronnen als een nieuw proces ingevoerd wordt, maar ook bij ieder verzoek om een hulpbron of bij ieder vrijkomen daarvan. De criteria voor het maken van dergelijke beslissingen en voor het geven van prioriteiten aan processen zijn meestal gebaseerd op ervaring; we bespreken er hieronder een aantal van.

(1) Processen die een groot aantal hulpbronnen bezitten krijgen een hoge prioriteit in een poging hun voltooiing te bespoedigen, zodat daardoor hulpbronnen terugverkregen worden die elders weer ingezet kunnen worden. Het gevaar dat in het volgen van deze strategie schuilt is, dat grote taken de machine alleen in beslag kunnen nemen of dat gebruikers 'nepverzoeken' voor hulpbronnen kunnen plaatsen, met het oog op het verkrijgen van een gunstiger behandeling. Dit laatste gevaar kan in een batchsysteem vermeden worden door ervoor te zorgen dat taken met een kleine behoefte aangaande de hulpbronnen bevoordeeld worden boven andere taken bij de selectie voor de uitvoering.
(2) Aan processen die een groot aantal hulpbronnen bezitten zouden verzoeken om meer hulpbronnen altijd toegewezen kunnen worden, als dat mogelijk is. De rechtvaardiging is dezelfde als hierboven en er is wederom het gevaar dat grote taken het systeem alleen in beslag nemen.
(3) De geheugentoewijzing in machines met een paginaverdeling kan in overeenstemming met het werkset-principe, dat in hoofdstuk 5 beschreven is, uitgevoerd worden.

(4) Processen van het besturingssysteem moeten een prioriteit hebben die in overeenstemming is met de urgentie van de functie die ze in het systeem uitvoeren. De schaal zal lopen van de werkindeler aan de top tot aan de processen, zoals die voor het dumpen van veranderde bestanden, onderaan. De meeste systeemprocessen zullen een hogere prioriteit hebben dan de gebruikersprocessen.
(5) Een speciaal geval van (4) is het geven van een hogere prioriteit aan de bestuurders van de randapparatuur. In het algemeen kan men stellen dat hoe sneller het apparaat is, des te hoger de prioriteit van de bijbehorende bestuurder moet zijn. Dit zorgt ervoor dat randapparatuur zoveel mogelijk aan de gang gehouden wordt, waardoor het gevaar voor kelpunten in de I/O verminderd wordt. Als er spooling gebruikt wordt, dan zouden de spoolers ook een hoge prioriteit moeten hebben.
(6) Tenzij er maatregelen voor het voorkómen van het vastlopen van kracht zijn, moeten alle verzoeken voor hulpbronnen onderworpen worden aan een algoritme voor het vermijden van het vastlopen.
(7) Het extra werk voor het maken van een werkindeling moet niet buiten proporties zijn ten opzichte van de eruit voortvloeiende voordelen.

De bovenstaande overwegingen kunnen gebruikt worden voor het veranderen van één van de eerder genoemde algoritmen voor het maken van een werkindeling voor de processor. Het resultaat zal een algoritme zijn waarin de prioriteiten van processen steeds worden bijgewerkt, in overeenstemming met het patroon van de verzoeken voor hulpbronnen en de toestand van het systeem. De prestaties van een dergelijk algoritme zijn moeilijk anders dan door waarneming te meten; de prestaties kunnen vaak aanzienlijk verbeterd worden door het aanbrengen van betrekkelijk geringe veranderingen in de parameters van het algoritme. Een interessant voorbeeld van een dergelijke verbetering kwam voor in het toezicht-houdende-programma (Engels: monitor) van de DEC System-10. In een van de eerste versies van het algoritme voor de werkindeling werden processen, die opnieuw in een uitvoerbare toestand kwamen nadat ze in afwachting van I/O met een invoerstation geblokkeerd waren, aan het einde van de eerste processorwachtrij geplaatst. Deze procedure werd vervolgens zo veranderd dat die processen aan het begin van de eerste wachtrij geplaatst werden. De verbetering in de responstijd voor de gebruikers van het meervoudig toegankelijke systeem was zeer aanzienlijk.

We kunnen deze sectie samenvatten door te zeggen dat het aantal mogelijke algoritmen voor de werkindeling erg groot is, maar dat we op het moment geen ander middel dan het experiment hebben voor het uitzoeken van een optimaal algoritme voor een bepaald geval. Analyse en simulatie kunnen soms helpen, maar hun waarde wordt beperkt door de complexiteit van de meeste systemen.

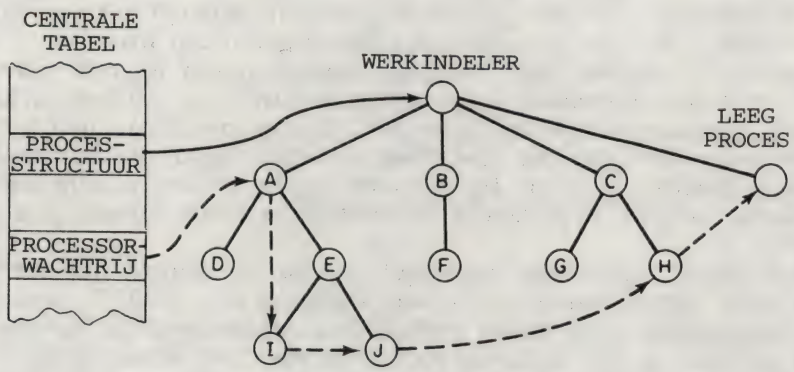
8.6 PROCES-HIËRARCHIEËN

We zagen in sectie 8.4 dat de werkindeler verantwoordelijk is voor het starten van nieuwe processen. Daarom is de werkindeler in een bepaald opzicht de vader van alle processen die in het systeem zijn ingevoerd. Temeer daar hij verantwoordelijk is voor het welzijn van zijn nakomelingen, vanwege het feit dat hij het beleid bepaalt voor de toewijzing van hulpbronnen en dat hij de volgorde beïnvloedt waarin de processen geselecteerd worden door het verdeelprogramma. Deze ouderlijke rol hoeft niet tot de werkindeler beperkt te zijn; deze kan uitgebreid worden door hem de mogelijkheid te geven tot

(1) het aanmaken van een subproces;
(2) het toewijzen van een eigen subset van hulpbronnen aan hun subprocessen (de hulpbronnen worden vrijgegeven als de subprocessen aflopen);
(3) het bepalen van de relatieve prioriteit van hun subprocessen.

Het resultaat is dat het maken van een proceshiërarchie met de werkindeler aan kop mogelijk wordt. Daardoor is de natuurlijke vorm van de processtructuur niet langer meer de lijst uit hoofdstuk 4, maar een boom zoals in figuur 8.5 weergegeven. De boom is gedeeltelijk van takken voorzien door de processorwachtrij, die alle knooppunten (processen) verbinden die op het moment uitvoerbaar zijn.

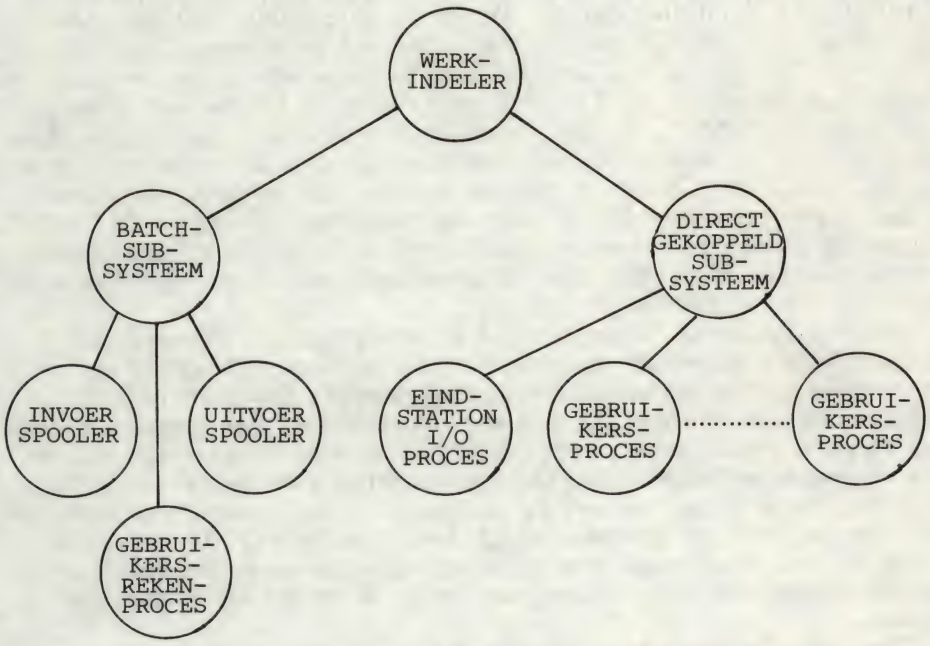
De voordelen van een hiërarchische oganisatie zijn tweeledig. Ten eerste stelt het een proces dat verschillende onafhankelijke taken uitvoert in staat een hulpproces te maken voor iedere taak, zodat de taken parallel uitgevoerd kunnen worden. Er wordt natuurlijk niets gewonnen in een machine met een enkele processor omdat er slechts schijngelijktijdigheid verkregen kan worden, maar als er meerdere processoren beschikbaar zijn, kan er werkelijk voordeel uit voortkomen.



Figuur 8.5 *Een proces-hiërarchie*

Ten tweede geeft het de mogelijkheid tot onafhankelijk en parallel lopen van meerdere versies van het besturingssysteem, elk toegespitst op een toepassing. De verschillende versies, of subsystemen, zullen alle gebaseerd zijn op de voorzieningen die door de systeemkern verzorgd worden (zie hoofdstuk 4), maar kunnen verschillende algoritmen gebruiken voor de implementatie van de hogere systeemfuncties die in de buitenste lagen verschijnen. Zodoende is ieder subsysteem een bepaalde versie van onze 'ui', alhoewel alle subsystemen dezelfde kern hebben. De diverse subsystemen implementeren elk een virtuele machine die geschikt is voor een bepaalde klasse gebruikers, en alle virtuele machines staan naast elkaar in dezelfde fysieke configuratie.

Laten we als voorbeeld eens figuur 8.6 bekijken die een situatie weergeeft waarin twee verschillende subsystemen verantwoordelijk zijn voor respectievelijk het batchwerk en het direct gekoppelde werk. De prestaties van de twee subsystemen worden bepaald door de hulpbronnen en de relatieve prioriteiten die door de werkindeler toegewezen zijn; binnen deze beperkingen kan ieder subsysteem zijn ondergeschikte processen in overeenstemming met elk geschikt algoritme besturen. Binnen het batch-subsysteem kan alle I/O bijvoorbeeld met behulp van spooling verwerkt worden en kunnen gebruikersprocessen geordend worden volgens het 'kortste taak eerst' principe; binnen het direct gekoppelde systeem kan de I/O zonder spooling gaan en kunnen de processen volgens het 'rondlopende wachtrij' principe geordend worden. Men kan zich een uitbreiding



Figuur 8.6 *Voorbeeld van meervoudige subsystemen*

van het hiërarchische systeem voorstellen waarin het batchsysteem bijvoorbeeld verder verdeeld wordt in subsystemen die de verschillende taakklassen afhandelen.

Een voordeel van de opstelling met subsystemen is het naast elkaar bestaan van verschillende virtuele machines. Een ander voordeel is de mogelijkheid tot het ontwikkelen van een nieuw besturingssysteem terwijl de oude versie er parallel aan loopt. Ook dit moet weer gesteld worden tegenover het verlies van het algemene beheer over de toewijzing van de hulpbronnen, als gevolg van de manier waarop hulpbronnen door de processtructuur heen en weer schuiven. Hulpbronnen die aan een subsysteem of aan een proces van ondergeschikt belang zijn toegewezen en die niet gebruikt worden, kunnen niet overgebracht worden naar een ander deel van het systeem, tenzij ze eerst teruggegeven worden aan een gemeenschappelijke voorouder. Omdat processen vrij zijn in het toewijzen van subsets van hun hulpbronnen, is er geen garantie dat het algemene patroon van het gebruik van de hulpbronnen in welk opzicht dan ook optimaal zal zijn.

De toewijzing van prioriteiten door een proces aan zijn afstammelingen kan ook problemen veroorzaken. Tenzij er een paar beperkingen worden opgelegd, is het mogelijk dat een proces op een oneerlijke manier een voordeel krijgt bij het maken van de werkindeling als het een taak naar een subproces verwijst waaraan het een erg hoge prioriteit toekent. Om dit probleem te voorkomen, wordt de eis gesteld dat alle prioriteiten toegewezen worden in overeenstemming met die van het ouderlijke proces en dat het bereik, waarover de relatieve prioriteiten mogen variëren, vermindert met elke stap naar beneden in de hiërarchie. Veronderstel bijvoorbeeld dat in figuur 8.5 het bereik van de relatieve prioriteiten op het hoogste niveau varieert van 0 - 99 en dat het bereik op het tweede niveau 0 - 9 is en zo verder. Als de prioriteit van proces B dan 20 is, dan mag geen van de subprocessen van B een prioriteit hebben die buiten het bereik 20 + 9 ligt. Dit betekent dat, als de prioriteit van proces A groter is dan 29, B geen voordeel ten opzichte van A kan krijgen door zich in subprocessen te splitsen. Als de prioriteit van A groter is dan 39, zullen alle subprocessen van A gegarandeerd de voorkeur hebben boven die van B.

In de situatie die in figuur 8.6 is weergegeven, bepalen de relatieve prioriteiten die door de werkindeler aan het batch- en direct gekoppelde subsysteem zijn toegewezen, daadwerkelijk de respons van het systeem op de twee taakklassen. Een groot verschil in prioriteiten tussen de subsystemen zal ervoor zorgen dat de ene gebruikersklasse altijd voorrang ten opzichte van de andere krijgt; een klein verschil zal tot gevolg hebben dat er op een gelijkwaardiger basis om de centrale processor(en) gevochten wordt. De relatieve prioriteit van een proces binnen ieder subsysteem is een kwestie die het betrokken subsysteem aangaat, zoals we al eerder opgemerkt hebben.

Een van de eerste besturingssystemen die een hiërarchische structuur vertoonden, was dat op de RC-4000 computer (Hansen,

1970). Naar de kern van dit systeem werd verwezen met de term 'Monitor' en de kop van de hiërarchie, die met onze werkindeler overeenkwam, werd het 'Basic Operating System' genoemd. Maximaal 23 processen (de beperking werd gemaakt vanwege de ruimte) werden van de ouders naar het nageslacht doorgegeven op de hierboven beschreven manier. Een verschil tussen de RC-4000 en het systeem dat wij beschreven hebben is, dat ouders geen prioriteiten aan hun kinderen konden toewijzen; alle processen hadden een gelijke prioriteit, behalve in het opzicht dat ze door het verdeelprogramma op basis van de rondlopende wachtrij werden geselecteerd.

Een beperkte hiërarchie is ook in het IBM VM/370 besturingssysteem (VM betekent Virtuele Machine) geïmplementeerd. De structuur is beperkt tot drie niveaus: op het hoofdniveau staat het VM/370 systeem zelf, op het volgende niveau kan er één voorkomen uit een aantal IBM standaard besturingssystemen (dat dienst doet als een subsysteem) en op het laagste niveau staan de gebruikersopdrachten. De standaard besturingssystemen die op het tweede niveau lopen voorzien in diverse virtuele machines op dezelfde configuratie. Het verdeelprogramma werkt volgens een principe met een rondlopende wachtrij op twee niveaus en kan gewijzigd worden door het toekennen van relatieve prioriteiten aan ieder subsysteem.

De implementatie van een hiërarchische processtructuur vereist het voorzien in de systeemprocedures:

maak proces (naam, relatieve prioriteit) *wis proces (naam)*
start proces (naam) *stop proces (naam)*

De functie van *maak proces* is het opstellen van een procesbeschrijver voor het nieuwe proces en het koppelen van die procesbeschrijver aan de processtructuur. *Start proces* voegt de procesbeschrijver toe aan de processorwachtrij, dat wil zeggen het markeert het proces als geschikt voor uitvoering. *Stop proces* en *wis proces* voeren de omgekeerde functies uit en geven de ouders de mogelijkheid hun kinderen te besturen. Kinderen kunnen natuurlijk op hun eigen houtje stoppen en het gebruik van *stop proces* is in dat geval niet nodig. Alle vier de procedures hebben een werking op de procesbeschrijvers of op de processorwachtrij en moeten daarom binnen in de systeemkern geïmplementeerd worden. De procedures *verzoek hulpbron* en *ontkoppel hulpbron* die in sectie 8.4 beschreven zijn, worden zo veranderd dat ze het ouderlijk proces aanroepen in plaats van de werkindeler. Het ouderlijke proces wordt op deze manier verantwoordelijk gemaakt voor de toewijzingen aan zijn nageslacht.

8.7 BESTURING EN VERANTWOORDING

De voorgaande secties hadden betrekking op de toewijzing van hulpbronnen in een tijdschaal die overeenkomt met de looptijden van taken. We verleggen onze aandacht nu naar de lange-termijnaspecten van de toewijzing van hulpbronnen, dat wil zeggen naar vragen als het in rekening brengen van het gebruik van hulpbronnen, het inschatten en beïnvloeden van het verzoekspatroon, en het ervoor zorgen dat alle gebruikers een gepast deel krijgen van de beschikbare computerfaciliteiten. Deze zaken worden meestal omschreven met de term *hulpbronbeheer*. Strikt genomen is het beheer van de hulpbronnen eerder een zorg voor de computerbeheerder dan voor de ontwerper van het besturingssysteem, maar als de ontwerper niet voor de juiste gereedschappen zorgt, staat de beheerder machteloos. De ontwerper moet in het bijzonder zorgen voor meetinstrumenten, want zonder metingen is beheer onmogelijk.

De oogmerken van hulpbronbeheer hangen af van de betrokken installatie. Men kan grofweg stellen dat er twee soorten installaties bestaan: *service* en *in-huis*. Het doel van een service-installatie, zoals die bestaan in servicebureaus, is het voorzien in computerfaciliteiten voor een aantal betalende gebruikers. Geen andere beperkingen dan de betalingsmogelijkheden worden de gebruikers opgelegd; het is in feite zelfs in het belang van de beheerders zoveel mogelijk faciliteiten aan zoveel mogelijk gebruikers te verkopen. Het beheer van de hulpbronnen wordt zo teruggebracht tot het *verantwoorden* van de hulpbronnen, dat wil zeggen het vastleggen van het gebruik van iedere hulpbron die berekend moet worden en het aan de hand daarvan sturen van rekeningen naar alle gebruikers.

Een installatie in-huis daarentegen is van een enkel bedrijf, universiteit of ander instituut en verstrekt alleen faciliteiten aan de leden van dat instituut. Vanwege de beperkingen die er aan zijn budget gesteld zijn, heeft de beheerder te maken met het probleem dat de vraag naar verwerking bijna altijd groter is dan de capaciteit van de installatie. In tegenstelling tot het servicebureau wordt er niets verdiend op het verlenen van de diensten en daarom bestaat de mogelijkheid van het vergroten van de capaciteit om aan de behoefte te voldoen meestal niet. De beheerder moet dus de beschikbare hulpbronnen in overeenstemming met de relatieve behoefte of belangrijkheid van de gebruikers rantsoeneren. Het beheer van de hulpbronnen voorziet in dit geval in een gereedschap voor het maken van onderscheid tussen de gebruikers. Het moet een dubbele functie uitvoeren, te weten: het verantwoorden van de hulpbronnen en het ervoor zorgen dat geen van de gebruikers over zijn rantsoen heen gaat.

Het zal duidelijk zijn dat de beheermechanismen die voor service-installaties nodig zijn een subset zijn van die, welke voor in-huis installaties nodig zijn. Daarom zullen wij ons op die laatste concentreren.

(1) Het verbieden van toegang

Toegang tot faciliteiten die een intensief gebruik van hulpbronnen met zich meebrengen kan ontzegd worden aan bepaalde gebruikersklassen, zoals jongste programmeurs of studenten. Ook hier weer kunnen gebruikers die via het meervoudige toegang systeem werken beperkt worden tot het op kleine schaal veranderen en testen, terwijl aan de gebruikers die via batchverwerking werken vrij gebruik van alle faciliteiten is toegestaan.

(2) Rantsoenering

Een beleid voor de rantsoenering kan voor de lange termijn, korte termijn of een combinatie van beide zijn. Het korte-termijnbeleid houdt in dat er een grens gesteld wordt aan de hoeveelheid gebruik die er van een hulpbron, zoals processortijd of geheugenruimte, gemaakt kan worden door een enkele taak. Bij het lange-termijnbeleid wordt een budget gegeven aan iedere gebruiker dat aangeeft hoeveel gebruik hij in een bepaalde periode, bijvoorbeeld een week of een maand, kan maken van hulpbronnen. Pogingen van de gebruiker over dit rantsoen heen te gaan hebben het afstoten van de taak, of de weigering van de toegang tot het systeem tot gevolg.

Beide methodes voor het beheer kunnen geïmplementeerd worden door het onderhouden van een verantwoordingsbestand, dat de volgende informatie bevat voor iedere gebruiker die tot het systeem is toegelaten.

(1) De identificatie van de gebruiker. Dit is een 'rekeningnummer' dat door de gebruiker opgegeven wordt als hij zich aanmeldt of bij het begin van een taakomschrijving. Het dient voor de identificatie van de gebruiker door het systeem.
(2) Een wachtwoord dat alleen aan de machine en de gebruiker bekend is. Dit wordt door de gebruiker getoond als bewijs van zijn identiteit. (Wachtwoorden zijn gewoonlijk niet vereist in batchsystemen omdat hun geheimhouding moeilijk te handhaven is als ze op een kaart of vergelijkbaar medium geponst worden.)
(3) Een 'toestemmingsvector' waarin iedere bit de toestemming voor het gebruik van een bepaalde faciliteit weergeeft. De toestemming wordt alleen gegeven als het bijbehorende bit is gezet.
(4) De grenzen voor het gebruik van hulpbronnen door een enkele taak.
(5) Het budget voor de huidige rekeningsperiode.
(6) De hulpbronnen die nog openstaan in het budget voor de huidige periode.

De details betreffende de toepassing van het beheer verschillen van systeem tot systeem. In batch-georiënteerde systemen wordt de

openstaande balans van de hulpbronnen aan het einde van iedere taak gedebiteerd en taken worden alleen door de werkindeler geaccepteerd als het saldo positief is. Een alternatief is het alleen accepteren van een taak als zijn vermelde behoefte aan hulpbronnen minder is dan het huidige saldo; dit voorkomt dat een gebruiker zijn rantsoen overschrijdt door het laten uitvoeren van een grote taak als zijn saldo bijna nul is. In een meervoudig toegankelijk systeem kan het saldo op regelmatige intervallen gedebiteerd worden gedurende een sessie aan het invoerstation en de gebruiker kan (na een duidelijke waarschuwing) gedwongen afgekoppeld worden als het saldo nul is geworden.

Rantsoenering kan op iedere hulpbron apart toegepast worden, of het gebruik van iedere hulpbron kan via de een of andere aangegeven berekeningsformule samengevoegd worden tot een soort universele 'hulpbron-verbruikseenheden'. In dat geval wordt het relatieve gewicht dat aan iedere hulpbron toegekend wordt in de berekeningsformule zo gekozen dat deze de belangrijkheid of schaarsheid van de hulpbronnen weergeeft. Een voorbeeld van een dergelijke berekeningsformule, dat uit de DEC System-10 van de universiteit van Essex stamt, is:

$$\text{aantal verbruikte eenheden} = (133{,}0P + 6{,}0C + 13{,}3K + 5{,}5W)$$

waarin:

P = de tijd van de centrale processor in seconden

C = de tijd dat het invoerstation verbonden is in seconden

K = geheugengebruik, zijnde de integraal over de processortijd van het aantal gebruikte, 1024 woorden grote, geheugenblokken

W = het aantal schijfoverdrachten

Merk op dat de parameters in een berekeningsformule gewijzigd kunnen worden om de werkwijze van de gebruikers te beïnvloeden. Zo zal een dramatische stijging van het tarief voor de verbinding van een invoerstation de gebruikers aanmoedigen meer gebruik van batchfaciliteiten te gaan maken, terwijl een stijging van het tarief voor het geheugengebruik een aansporing tot het schrijven van kleine taken kan zijn.

Een ander belangrijk kenmerk van een rantsoeneringsbeleid is, dat de totale toegewezen hoeveelheid van iedere hulpbron gerelateerd zou moeten zijn aan de hoeveelheid van die hulpbron waarin het systeem in een gegeven periode kan voorzien. De toegewezen hoeveelheid hoeft niet gelijk te zijn aan die waarin voorzien is, omdat er een gerede kans bestaat dat de meeste gebruikers niet hun gehele toewijzing zullen verbruiken; de veilige mate van overtoewijzing kan uit ervaring afgeleid worden. In het geval van een universele berekeningsformule zijn de toewijzing en aanvoer moeilijker met elkaar

in overeenstemming te brengen, want als alle gebruikers beslissen hun toewijzing in de vorm van dezelfde hulpbronnen te gebruiken, zou het systeem dat niet aankunnen. In de praktijk maakt de aanwezigheid van een grote gebruikersgroep deze eventualiteit onwaarschijnlijk.

Meer verfijnde beleidsvormen voor de rantsoenering (bijvoorbeeld Hartley, 1970) kunnen gebruik maken van verschillende tarieven op verschillende tijdstippen van de dag. Dit moedigt het gebruik van de machine op impopulaire tijden aan en geeft een gelijkmatiger verdeling van de werklast. Op dezelfde manier kunnen de tarieven ook variëren naar gelang de externe prioriteit die de gebruiker voor zijn werk heeft aangevraagd.

Het is de verantwoordelijkheid van de systeemontwerper de hulpmiddelen te verstrekken voor het implementeren van beleidsvormen zoals die welke zojuist beschreven zijn. Deze hulpmiddelen omvatten:

(1) routines voor het vastleggen van het bestand waarin de verantwoording staat;
(2) mechanismen voor het meten van het gebruik van hulpbronnen;
(3) mechanismen voor het weigeren van faciliteiten aan gebruikers die over hun budget zijn heen gegaan.

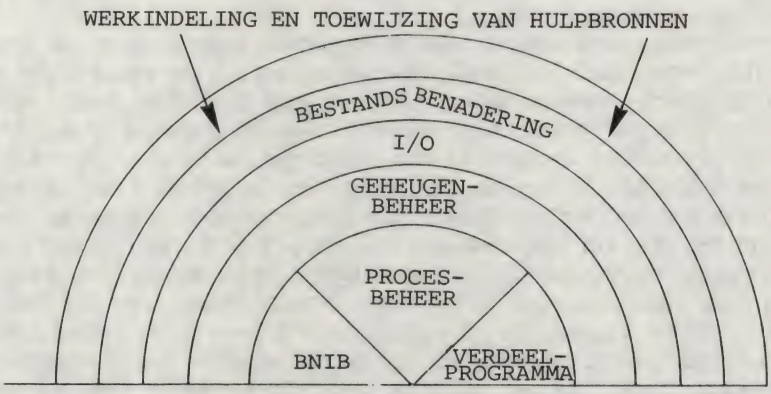
Het meten van het gebruik van hulpbronnen is een functie die door het hele systeem verspreid is op punten waar de hulpbronnen toegewezen worden. Het kan aanzienlijk verlicht worden door het voorzien in de juiste hardware, zoals een teller in schijfbesturing voor het vastleggen van het aantal overdrachten. Een real-time klok is natuurlijk essentieel voor nauwkeurige metingen. Metingen betreffende een individueel proces kunnen als deel van de procesbeschrijver opgeslagen worden en statistische gegevens aangaande het gehele systeem kunnen verzameld worden in een centraal bestand. Die laatste getallen kunnen beschouwd worden als een indicatie over de prestaties van het systeem en ze kunnen gebruikt worden als een basis voor vergelijking en verbetering.

Er moet misschien ook aandacht gegeven worden aan de vraag of het extra werk van het besturingssysteem bij de gebruiker in rekening gebracht moet worden. Aan de ene kant kan men stellen dat het besturingssysteem ten behoeve van de gebruiker werkt en dat het de gebruiker daarom in rekening gebracht moet worden. Dit zou de gunstige bijwerking hebben dat de gebruikers aangemoedigd worden tot het zó schrijven van programma's dat ze zo min mogelijk extra last op het systeem leggen, door bijvoorbeeld overtollige toegang tot bestanden te vermijden. Aan de andere kant zal de extra last variëren met de huidige bezetting en men zou het oneerlijk kunnen vinden dat de gebruiker alleen meer berekend wordt omdat de machine bezig is, of omdat zijn taak een ongunstige wisselwerking heeft met een andere taak die toevallig tegelijkertijd in het systeem aanwezig is. Men kan ook redeneren dat er van de gewone gebruiker niet verwacht kan worden dat hij de gedetailleerde

kennis verwerft over het besturingssysteem die hij nodig zou hebben voor het zodanig veranderen van zijn werk dat de extra last verminderd. Een mogelijk compromis is het aan de gebruiker in rekening brengen van alle systeemactiviteiten, zoals I/O verwerking, die specifiek voor hem verricht worden, maar het niet in rekening brengen van activiteiten, zoals het schakelen tussen processen, die buiten zijn macht liggen. Dit laat echter nog een aantal gebieden voor discussie open; het verkeer ten gevolge van het verwisselen van pagina's is bijvoorbeeld afhankelijk van zowel de gebruiker, die het gebied van zijn verwijzingen naar het geheugen kan vergroten, als van de huidige omvang van de vraag naar geheugenruimte. Uiteindelijk zijn beslissingen op dit gebied noodgedwongen arbitrair en moeten aan de individuele ontwerper overgelaten worden.

8.8 SAMENVATTING

In dit hoofdstuk hebben we gezien hoe beleidsvormen voor het toewijzen van hulpbronnen en het maken van een werkindeling in een enkel proces geïmplementeerd kunnen worden; dat proces hebben we de werkindeler genoemd. We hebben beschreven hoe de functies van de werkindeler uitgebreid kunnen worden naar andere processen, waarmee een hiërarchische processtructuur gevestigd wordt. De kern van ons papieren besturingssysteem is vergroot doordat we er procedures voor het aanmaken en wissen van processen in opgenomen hebben. De mechanismen voor het toewijzen van hulpbronnen en voor het meten van het gebruik van hulpbronnen zijn over het hele systeem verspreid op de niveaus die bij de betrokken hulpbronnen horen. De huidige toestand van ons systeem is in figuur 8.7 weergegeven.



Figuur 8.7 *De huidige toestand van het papieren besturingssysteem*

De toegevoegde gegevensstructuren zijn:

(1) een lijst van beschikbare hulpbronnen;
(2) voor ieder proces een lijst van de eraan toegewezen hulpbronnen, waarnaar verwezen wordt vanuit de procesbeschrijver;
(3) als herkenning van het vastlopen gebruikt wordt, dan een weergave van de toestandssituatie;
(4) als vermijding van het vastlopen gebruikt wordt, dan een lijst van de claims en toewijzingen;
(5) een mogelijke uitbreiding van de processorwachtrij zodat deze meerdere niveaus omvat;
(6) een mogelijke wijziging van de processtructuur zodat die boomvormig wordt;
(7) een verantwoordingsbestand.

9 Bescherming

We hebben al een aantal malen gezinspeeld op het feit dat de processen in een computersysteem tegen elkaars activiteiten beschermd moeten worden. We hebben op diverse punten in ons papieren besturingssysteem mechanismen geïntroduceerd die ervoor zorgen dat het geheugen, de bestanden en de hulpbronnen die bij een proces horen, niet door een ander proces benaderd kunnen worden, behalve op een ordelijke manier. De mechanismen zijn over het systeem verdeeld en aangebracht op plaatsen zoals de hardware voor de geheugenadressering, bestandsindexen en toewijzingsprocedures voor hulpbronnen, en ze functioneren min of meer onafhankelijk van elkaar. In dit hoofdstuk bekijken we het onderwerp 'bescherming' in zijn totaliteit nader en geven aan hoe het mogelijk is beschermingsmechanismen te implementeren op een uniforme wijze in het gehele systeem.

9.1 BEWEEGREDENEN

De beweegredenen om beschermingsmechanismen in te bouwen in een computersysteem, dat voor meerdere gebruikers bedoeld is, kunnen als volgt samengevat worden.

(1) Bescherming tegen storingen

Als er meerdere processen, die mogelijk voor meerdere gebruikers werken, op dezelfde machine uitgevoerd worden, is het duidelijk ongewenst als storingen in één van de processen de uitvoering van de andere processen beïnvloeden. Er zijn maar weinig gebruikers die een systeem tolereren waarin ieder proces gegarandeerd storingsvrij moet zijn, vóórdat ze er zeker van kunnen zijn dat hun eigen werk zonder problemen zal verlopen. Daarom moeten er mechanismen ontworpen worden voor het oprichten van 'brandmuren' tussen de verschillende processen in het systeem. De bescherming tegen storingen slaat niet alleen op gebruikersprocessen maar ook op het besturingssysteem zelf. Als het besturingssysteem in de war geschopt kan worden door storingen in een gebruikersproces, zal het waarschijnlijk niet lang (of helemaal niet) goed functioneren.

Bescherming is zelfs in het geval van een gebruiker met zijn eigen machine wenselijk. In deze situatie kan de bescherming een hulp bij het zoeken naar fouten zijn, doordat deze de voortzetting van fouten beperkt en door het aangeven van de oorspronkelijke plaats van de fout. Zelfs als alle fouten uit de programma's gehaald zijn, kan bescherming de schade minimaliseren die veroorzaakt wordt door het niet goed werken van de hardware of door fouten van degene die de computer bedient. Deze overwegingen zijn nog sterker van toepassing bij systemen voor meerdere gebruikers.

(2) Bescherming tegen kwaadwilligheid

In een tijd waarin computers steeds meer gebruikt gaan worden voor het opslaan en bewerken van gegevensbestanden, waarvan vele vertrouwelijke informatie bevatten, is het essentieel dat de veiligheid van de gegevens verzekerd is. Die zekerheid kan wegens commerciële redenen nodig zijn, zoals in het geval van een installatie die diensten verzorgd voor gebruikers met strijdige belangen, of het kan vereist zijn om redenen van privacy. Medewerkers van overheidsinstanties, zoals bij het bevolkingsregister of de belastingdienst, mogen bijvoorbeeld niet zonder toestemming toegang kunnen krijgen tot vertrouwelijke persoonsgegevens. De veiligheidsmaatregelen, die voor gegevens nodig zijn, gaan net zo goed op voor programma's. Zo zal bijvoorbeeld een bedrijf dat veel geld in de ontwikkeling van een programma gestoken heeft dat het wil gaan verkopen of verhuren, niet erg blij zijn als gebruikers of concurrenten het programma kunnen kopiëren en elders kunnen installeren. Vergelijkbaar hiermee zou de belastingdienst behoorlijk vreemd opkijken als ze zou merken dat een van haar medewerkers een programma zo gewijzigd had dat hij al vijf jaar geen belasting betaald heeft.

Een besturingssysteem is zelf een verzameling programma's en gegevens die tegen misbruik beschermd moet worden. Het is niet moeilijk in te denken dat een gebruiker een onbeschermd systeem zo wijzigt dat zijn taken altijd als eerste ingedeeld worden, of dat hij nooit de hulpbronnen die hij gebruikt in rekening gebracht krijgt.

Als bescherming alleen maar een kwestie zou zijn van het onderling isoleren van processen, dan zouden adequate mechanismen in een handomdraai ontworpen kunnen worden. Helaas kunnen processen gegronde redenen hebben voor de volgende wensen: gedeeld gebruik kunnen maken van hulpbronnen, toegang hebben tot gemeenschappelijke gegevens of dezelfde programma's kunnen laten uitvoeren. Een waardevol beschermingssysteem moet daarom legitiem deelgebruik toestaan, terwijl het niet geautoriseerde inbreuken tegengaat. Het naast elkaar in het computersysteem bestaan van de elementen samenwerking en onderlinge competitie, verzoorzaakt de problemen voor de bescherming.

9.2 ONTWIKKELING VAN BESCHERMINGSMECHANISMEN

Bescherming in de elementaire betekenis: het gehele systeem uitsluiten voor niet geautoriseerde gebruikers, kan bereikt worden door aan de gebruiker een bewijs te vragen voor zijn identiteit, als hij in het systeem probeert te komen of als hij een taak opgeeft. De gebruiker doet dat meestal door het geven van zijn rekeningnummer, dat gebruikt wordt voor het in rekening brengen van de gebruikte hulpbronnen, en door het geven van een wachtwoord dat alleen aan hem en het besturingssysteem bekend is. Het rekeningnummer en het wachtwoord worden gecontroleerd aan de hand van het verantwoordingsbestand, zoals in sectie 8.7 is beschreven. Op voorwaarde dat het verantwoordingsbestand ontoegankelijk is voor gebruikers en op voorwaarde dat de gebruikers hun wachtwoord geheim houden, kunnen alleen geautoriseerde gebruikers toegang krijgen tot het systeem. Helaas is het laatste voorbehoud moeilijk in de praktijk af te dwingen omdat gebruikers de gewoonte hebben hun wachtwoorden door onoplettendheid, of door een misplaatst gevoel van edelmoedigheid, vrij te geven. Speciaal in batchsystemen is het bijzonder moeilijk wachtwoorden geheim te houden, omdat ze op kaarten of een vergelijkbaar medium geponst moeten worden. Om die reden vragen de meeste batchsystemen helemaal geen wachtwoord. Zelfs bij meervoudig toegankelijke systemen, waarbij de gebruikers strikte geheimhouding betrachten, is aangetoond dat een vastbesloten indringer door gericht proefondervindelijk te werken uiteindelijk een geldig wachtwoord kan ontdekken en toegang tot het systeem kan krijgen. We zullen dit onderwerp hier niet verder bespreken, maar we zullen ons in de rest van dit hoofdstuk concentreren op de beschermingsmaatregelen die toegepast kunnen worden nadat de gebruikers (al dan niet legitiem) toegang tot het systeem hebben gekregen.

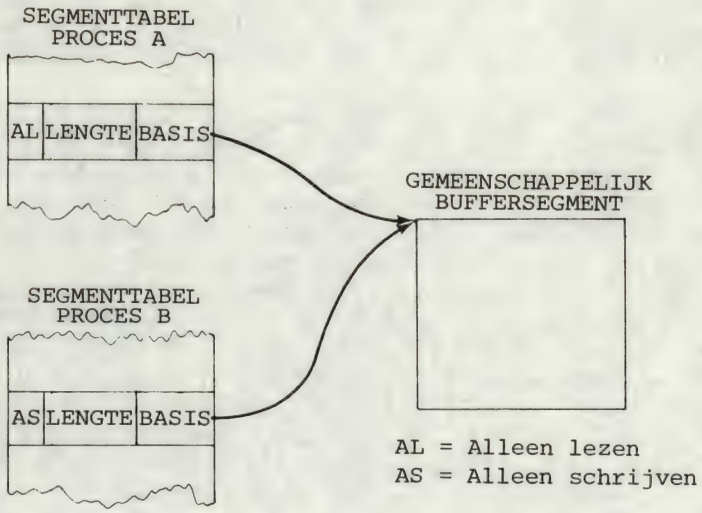
Een rudimentaire vorm van bescherming kan verkregen worden door het opnemen van een *toestandsschakelaar* en een paar *basisgrensregisters* in de hardware van de moedermachine. De toestandsschakelaar regelt de mogelijkheid tot het uitvoeren van de bevoorrechte instructies zoals die beschreven zijn in sectie 4.1. De basisgrensregisters geven in de gebruikerstoestand het voor een proces toegankelijke geheugengebied aan (zie sectie 5.3). Informatie die in het hoofdgeheugen aanwezig is wordt beschermd door de basis- en grensregisters, waarvan de waarden alleen in de supervisortoestand veranderd kunnen worden; informatie op het achtergrondgeheugen wordt beschermd dankzij het feit dat I/O instructies bevoorrecht zijn en dat zodoende alle gegevensoverdrachten alleen via het besturingssysteem kunnen plaatsvinden. Toestemming voor de overdrachten kan verkregen worden door het gebruik van de informatie die in de bestandsindexen staat wanneer de bestanden geopend en gesloten worden, zoals in hoofdstuk 7 is beschreven.

Deze beschermingsvorm wordt in de meeste computers uit het midden van de jaren '60 gevonden (bijvoorbeeld de ICL 1900 serie). Het is onbevredigend dat het een 'alles of niets' oplossing is voor

het beschermingsprobleem: een proces heeft òf geen voorrechten (als het in de gebruikerstoestand loopt) òf heeft alle voorrechten (als het in de supervisotoestand loopt). Zodoende hebben bijvoorbeeld processen die een willekeurige systeemprocedure uitvoeren dezelfde absolute voorrechten, onafhankelijk van het feit of zij ze nodig hebben. Verder is gemeenschappelijk gebruik van gegevens of programma's alleen mogelijk door het laten overlappen van hun geheugenruimtes; in dat geval zijn processen ten opzichte van elkaar volledig onbeschermd.

Een flexibeler beschermingsschema kan worden geïmplementeerd als de moedermachine voorziet in segmentatiehardware (zie sectie 5.3). In dat geval wordt de adresruimte van elk proces in logische segmenten ingedeeld en elk segment kan een bepaalde mate van bescherming toegewezen krijgen (zoals: alleen uitvoeren, alleen lezen) door het inbouwen van een beschermingsindicator in zijn beschrijver. De situatie is in figuur 5.5 weergegeven. De hardware voor de geheugenadressering controleert alle toegang tot de segmenten om te verzekeren dat de in de beschrijver opgegeven bescherming niet geschonden wordt.

Omdat ieder proces door middel van zijn eigen set segmentbeschrijvers (opgeslagen in zijn segmenttabel) naar zijn segmenten verwijst, is het mogelijk dat twee processen naar hetzelfde fysieke segment verwijzen met behulp van beschrijvers die andere beschermingsindicaties hebben. Dit maakt het mogelijk dat segmenten gemeenschappelijk gebruikt worden door processen met verschillende toegangsrechten. Figuur 9.1 geeft een voorbeeld waarin proces *A* informatie leest die door proces *B* in een buffer is gedeponeerd. De



Figuur 9.1 *Gemeenschappelijk segmentgebruik met verschillende toegangsrechten*

buffer verschijnt als een segment in de adresruimte van zowel *A* als *B*; zijn beschrijver in proces *A* geeft toegang aan voor alleen lezen, terwijl zijn beschrijver in proces *B* toegang voor schrijven toestaat. Beide beschrijvers verwijzen naar dezelfde plaats in het geheugen.

Alhoewel dit schema een verbetering is op het vorige, geeft het ons nog steeds niet afdoende bescherming. De reden hiervoor is dat de toegangsrechten die een proces heeft gedurende zijn gehele bestaan onveranderd blijven. Dit betekent dat het proces, onafhankelijk van welk programma of welke procedure het ook uitvoert, dezelfde voorrechten handhaaft voor al zijn segmenten. (Een uitzondering hierop is die waarbij het proces de supervisortoestand ingaat; hierin gaat het uit van absolute voorrechten voor alle segmenten.) Een opstelling die de voorkeur verdient, maar waarin door het bovenstaande schema niet wordt voorzien, is die waarbij een proces te allen tijde alleen die voorrechten bezit die het nodig heeft voor de werkzaamheden waar het op dat moment mee bezig is. Deze filosofie van 'minimaal noodzakelijke voorrechten' is noodzakelijk om de gevolgen van foutieve of kwaadwillende processen te beperken.

Laten we als voorbeeld nog eens naar de processen *A* en *B* uit figuur 9.1 kijken. Als de buffer wordt uitgebreid met communicatie in twee-richtingsverkeer, dan moeten zowel *A* als *B* schrijftoegang hebben tot de buffer. Dit betekent dat elk proces de mogelijkheid heeft naar de buffer te schrijven; zodoende kan de informatie in de buffer niet gegarandeerd beschermd worden tegen het verkeerd werken van een van de twee processen. Het is noodzakelijk dat de toegangsvoorrechten van *A* en *B* voor de buffer afhankelijk zijn van het feit of zij op het moment bezig zijn met het zenden of met het ontvangen van informatie. Zodoende moet, als één van de twee processen bezig is met een procedure voor het zenden van informatie, dat proces schrijftoegang tot de buffer hebben, maar als het bezig is met een procedure voor het lezen, dan moet het alleen leestoegang hebben.

Als een gevolg van deze bespreking introduceren we het idee van een beschermings*domein* (andere omschrijvingen zijn: *sfeer*, *context* en *regime*) waarin een proces loopt. Ieder domein definieert een set voorrechten of *functies* die uitgeoefend kunnen worden door een proces dat erbinnen wordt uitgevoerd. Een verandering in de functies van het proces kan alleen bereikt worden door het over te brengen van het ene naar het andere domein; we staan er natuurlijk op dat zulke overdrachten bijzonder goed gecontroleerd worden.

Primitieve voorbeelden van domeinen zijn de supervisor- en gebruikers-uitvoeringstoestand die eerder vermeld zijn. Als een proces in de supervisortoestand werkt, behoren de uitvoering van bevoorrechte instructies en de verwijzing naar het gehele geheugen tot de functies die hij kan gebruiken; als het in de gebruikerstoestand is, zijn deze functies beperkt tot het gebruik van niet bevoorrechte instructies en tot de toegang van alleen zijn eigen geheugenruimte. Vanwege de eerder besproken redenen is het twee-domeinensysteem onvoldoende voor onze doeleinden: we hebben een complexere opstelling in gedachten, waarin een aantal domeinen een grote variëteit

aan functies definieert en waarin een proces werkt in het domein dat precies die functies heeft die het in staat stellen tot het verrichten van zijn legitieme werk. We zullen in de volgende secties onderzoeken hoe een dergelijke opstelling geïmplementeerd kan worden.

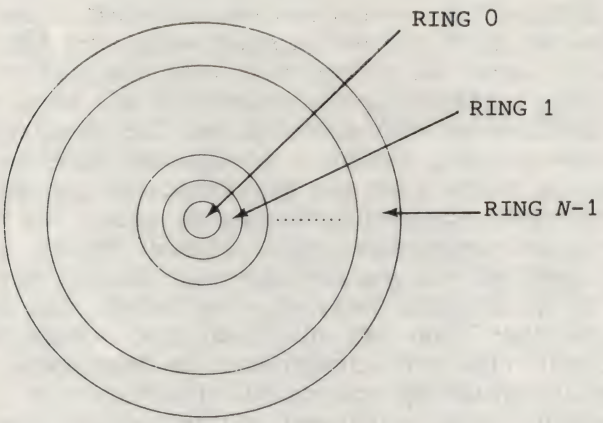
Merk op dat functies voor alle objecten, die bescherming vereisen, uitgedrukt kunnen worden in termen van functies voor geheugensegmenten. De toegang tot een gegevensstructuur van het systeem kan bijvoorbeeld uitgedrukt worden in termen van toegang tot het segment dat deze gegevensstructuur bevat, terwijl de toestemming voor het gebruik van een randapparaat afhangt van de toegang tot het segment dat de apparaatbeschrijver bevat. (In sommige computers, zoals de PDP-11 en vele microcomputers, staan de apparatuuradressen in de algemene adresruimte; in deze computers kan de toegang tot de apparaten op precies dezelfde manier bestuurd worden als de toegang tot het geheugen.) Deze constatering leidt tot een aanzienlijke vereenvoudiging van beschermingsschema's, omdat de bescherming nu uitgevoerd kan worden door de segmentfuncties, die op hun beurt weer als een deel van de mechanismen voor de geheugenadressering geïmplementeerd kunnen worden. Deze benadering is in de hieronder beschreven schema's gevolgd.

9.3 EEN HIËRARCHISCH BESCHERMINGSSYSTEEM

In deze sectie beschrijven we het beschermingsschema dat in MULTICS (Graham, 1968; Schroeder en Saltzer, 1972) is geïmplementeerd. MULTICS was een van de eerste systemen waarin de bescherming op een algemene manier werd behandeld en het is een inspiratiebron geweest voor systemen als VME op de ICL 2900 en NOS/VE op de CYBER 180. Onze beschrijving is wat vereenvoudigd, maar voldoet ter illustratie van de ideeën achter het schema van een hiërarchisch georganiseerde bescherming.

De domeinen in MULTICS worden *ringen* genoemd en ze zijn zo geordend dat iedere ring een subset bevat van de functies van de erbinnen liggende ring. Men kan zich het principe ervan voorstellen zoals in figuur 9.2 weergegeven, waarin de voortgang van de buitenste (hoogst genummerde) naar de binnenste (laagst genummerde) ringen steeds grotere voorrechten inhouden. De lezer zal begrijpen dat dit een verfijning is van de supervisor/gebruiker beschermingsmethode, die in feite als een twee-ringensysteem beschouwd kan worden.

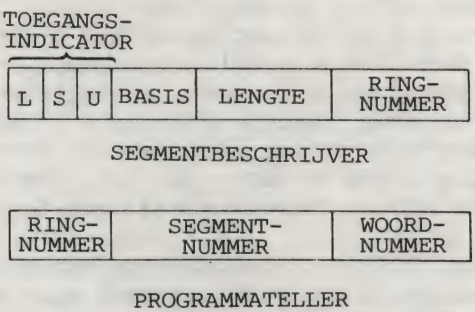
Een ring is in MULTICS een verzameling segmenten. Voor het moment zullen we aannemen dat ieder segment maar aan één enkele ring is toegewezen. Procedures, die een groot aantal voorrechten nodig hebben, zetelen in de binnenste ringen, terwijl procedures, die minder privileges nodig hebben of waarvan men niet zeker is dat ze vrij van fouten zijn, in de buitenste ringen opgeslagen zijn.



Figuur 9.2 *Beschermingsringen*

Voor het aangeven van de ring waartoe een segment behoort wordt de segmentbeschrijver uitgebreid, zodat het een veld bevat waarin het ringnummer staat. De programmateller wordt op een overeenkomstige manier uitgebreid voor het aangeven van de ring waarin het huidige proces loopt (dat wil zeggen het ringnummer van het segment waaruit het instructies uitvoert). De indicatie voor de bescherming in de segmentbeschrijver omvat vlaggen die aangeven of een segment beschreven, gelezen of uitgevoerd kan worden (zie figuur 9.3).

Een proces dat in ring i uitgevoerd wordt, heeft geen enkele toegang tot de segmenten in ring j, waarbij $j < i$; zijn toegang tot de segmenten in ring k, waarbij $k \geq i$, wordt geregeld door de toegangsindicatoren van de betrokken segmenten. Met andere woorden:



Figuur 9.3 *Segmentbeschrijver en programmateller van MULTICS*

toegang tot de erbinnen liggende ringen is verboden, terwijl toegang tot de erbuiten liggende ringen afhangt van de toegangsindicatoren. Pogingen van een proces om een grens tussen ringen over te gaan door het aanroepen van een procedure in een andere ring, zullen uitmonden in een interne foutmelding. De foutmelding resulteert in het binnengaan van een speciale foutbehandelingsprocedure die bepaalde controles uitvoert op de overdracht van de besturing. Deze controles zullen zo meteen besproken worden.

Het hierboven beschreven mechanisme is afdoende voor het voorzien in een hiërarchische bescherming. Het feit dat een segment aan slechts één ring is toegewezen leidt echter tot verspilling als gevolg van het grote aantal procedures aan de andere zijde van een ringgrens dat aangeroepen wordt. Een meer flexibele opstelling, die in MULTICS is toegepast, is het toewijzen van ieder segment aan een stel opeenvolgende ringen, het *toegangsbereik* genaamd. Het aanroepen van een procedure in een segment met toegangsbereik (n_1,n_2) door een proces dat in ring i werkt, veroorzaakt nu geen foutmelding op de voorwaarde dat $n_1 \leqq i \leqq n_2$ en in dit geval blijft de besturing in ring i.

Er wordt een foutmelding veroorzaakt als $i > n_2$ of als $i < n_1$ en de bijbehorende besturing moet beslissen of het aanroepen toegestaan moet worden. (De foutbesturing kan een procedure in ring 0 zijn of kan, zoals in MULTICS, voor een groot deel geïntegreerd zijn in de hardware voor de geheugenadressering.) Het geval waarin $i < n_1$ geeft het aanroepen naar buiten weer, dat wil zeggen de overdracht van de besturing naar een ring met minder voorrechten, en kan daarom altijd worden toegestaan. De parameters van het aanroepen kunnen echter verwijzen naar segmenten die, vanwege hun lagere status, ontoegankelijk zijn voor de aangeroepen procedure. In dat geval moeten de parameters gekopieerd worden naar een gegevensgebied dat wel toegankelijk is voor de aangeroepen procedure.

Het aanroepen in de tegenovergestelde richting ($i > n_2$) is moeilijker te verwerken. Omdat het aanroepen een overdracht naar een ring met meer voorrechten inhoudt, moet men oppassen dat de aanroeper er op geen enkele wijze de oorzaak van kan zijn dat de aangeroepen procedure niet goed meer functioneert. Een eerste stap in die richting is het alleen toestaan van aanroepen die niet te ver af liggen van het toegangsbereik van de aangeroepen procedure. Een gehele en positieve waarde n_3 ($n_3 > n_2$) die in iedere segmentbeschrijver staat, wordt gebruikt voor het aangeven van de grenzen van het 'aanroepbereik' van het segment; een aanroep vanuit ring i waarbij $i > n_3$, is niet toegestaan. Verder bezit ieder segment een lijst (die leeg kan zijn) van de ingangspunten of *poorten* (Engels: *gates*) via welke het aangeroepen kan worden. De foutbesturing bekijkt of alle binnen gelegen aanroepen voor procedures zijn gericht op een van deze poorten en veroorzaakt een foutmelding als dat niet zo is. De foutbesturing legt stapelsgewijs het terugkeerpunt en het ringnummer vast die bij de aanroep horen. Een terugkeer over een ring wordt vergeleken met de bovenste gegevens van de stapel

om ervoor te zorgen dat een procedure niet op slinkse wijze terugkeert naar een lagere ring dan die waaruit hij vertrok. De parameters van een naar binnen gerichte aanroep moeten ook op geldigheid gecontroleerd worden, om te voorkomen dat zij verwijzen naar segmenten die tegen de aanroeper beschermd zouden moeten zijn.

Hiërarchische beschermingssystemen zoals hierboven beschreven hebben het nadeel dat, als een object wel toegankelijk moet zijn vanuit domein *A* maar niet vanuit domein *B*, *A* dan hoger in de hiërarchie moet staan dan *B*. Dit houdt in dat alles wat vanuit *B* toegankelijk is noodzakelijkerwijs ook vanuit *A* toegankelijk moet zijn. Hiermee wordt het doel om aan een proces te allen tijde de minimum noodzakelijke voorrechten te geven volkomen gemist. De diverse beschermingslagen weerspiegelen echter in een bepaald opzicht de bouwlagen in een besturingssysteem en vormen daarom van nature een aantrekkelijke manier voor het versterken van de structuur van dat besturingssysteem. De ontwerpers van het MULTICS systeem pretenderen dat de hiërarchische structuur in het gebruik geen hinderlijke beperkingen oplegt en dat het kan voorzien in een bevredigende bescherming. In de volgende sectie zullen we een alternatief bekijken dat niet afhankelijk is van een hiërarchische organisatie.

9.4 ALGEMENE SYSTEMEN

In het hierboven beschreven schema voor hiërarchische bescherming zijn de functies, die een proces tot zijn beschikking heeft, direct afhankelijk van het ringnummer waarin het proces loopt. Tevens houdt de overdracht van de besturing van de ene naar de andere ring een uitbreiding of beperking in van de op dat moment beschikbare functie. In een niet hiërarchisch systeem kan de overdracht van de besturing tussen domeinen echter gepaard gaan met volkomen willekeurige veranderingen in de voor een proces beschikbare functies. De set functies die bij een proces in een bepaald domein horen, heet zijn *functielijst*. In een hiërarchisch systeem is de nieuwe functielijst altijd een uitbreiding of subset van de oude, terwijl er in een niet hiërarchisch systeem geen duidelijk verband is.

Een abstract model voor een algemeen beschermingssysteem (Graham en Denning, 1972) is een *toegangsmatrix* (figuur 9.4) die de functies aangeeft voor de diverse *onderdanen* voor alle systeem-*onderdelen* die bescherming nodig hebben. Een onderdaan is een paar (*proces, domein*), elke rij in de matrix geeft dus de lijst van functies voor een bepaald proces aan, dat in een bepaald domein loopt. Als een proces van domein verandert, wordt het een andere onderdaan en wordt zijn lijst van functies door een andere rij uit de toegangsmatrix aangegeven. In het algemeen vereisen onderdanen bescherming tegen elkaar, daarom worden onderdanen ook als onderdelen beschouwd.

ONDERDANEN	ONDERDELEN						
	Onderdanen			Bestanden		Apparaten	
	O_1	O_2	O_3	B_1	B_2	A_1	A_2
O_1		stop wis		lees schrijf			zoek
O_2			stop		werk bij	schrijf	
O_3				wis	voer uit		

Figuur 9.4 *Deel van een matrix voor de toegangsbesturing*

Aan ieder onderdeel is een *besturingseenheid* (Engels: *controller*) verbonden, die steeds ingeschakeld wordt wanneer het onderdeel benaderd wordt. De besturingseenheid kan, afhankelijk van het betrokken onderdeel, in de hard- of software geïmplementeerd zijn; een typerend voorbeeld zou het onderstaande kunnen zijn:

Soort onderdeel	Besturingseenheid
bestand	opslagsysteem
segment	hardware voor de adressering van het geheugen
randapparaat	apparaatbesturing

De besturingseenheid is verantwoordelijk voor de controle of een onderdaan de juiste functie bezit voor het gevraagde soort toegang. Een bepaald beschermingsschema wordt gekenmerkt door de keuze van de onderdanen en de onderdelen, door de functies die aangegeven worden door de elementen in de toegangsmatrix, en door de manier waarop de functies veranderd of overgedragen kunnen worden.

Het model bevat alle onderdelen van een algemeen beschermingssysteem, maar een efficiënte implementatie voor een groot aantal soorten onderdelen is moeilijk voor te stellen. Als echter, zoals in sectie 9.2 voorgesteld, het aantal onderdelen beperkt wordt tot geheugensegmenten en als een proces alleen van domein mag veranderen wanneer het een nieuw segment binnengaat, dan kan de

implementatie bereikt worden met behulp van de hardware voor de geheugenadressering. Twee benaderingen voor een dergelijke implementatie worden hieronder beschreven.

De eerste benadering (Evans en Leclerc, 1967) is die waarbij een functie van de vorm (*toegangsindicatie, segmentbasis, segmentlengte*) als identiek beschouwd wordt aan een segmentbeschrijver, en waarbij de lijst van functies van een proces met zijn segmenttabel geïdentificeerd wordt. Als een proces van domein verandert, houdt dat een verandering van de segmenttabel in. Een gevolg van deze benadering is de vernietiging van het begrip 'proces-brede adresruimte', omdat een proces in overeenstemming met het domein waarin het loopt van adresruimte verandert. Er worden ook enige moeilijkheden ondervonden bij het doorgeven van de parameters: een recursieve methode vereist een aparte segmenttabel voor ieder aanroepniveau, waarbij de opeenvolgende tabellen toegang geven tot de bij ieder niveau behorende parameters.

Een alternatieve benadering is die welke in de Plessey 250 computer (Williams, 1972; Englang, 1974) toegepast is. We zullen deze implementatie wat nader beschrijven, omdat het de manier weergeeft waarop schema's voor geheugenadressering, die op 'functies' gebaseerd zijn, tot op zekere hoogte de traditionelere schema's, die in hoofdstuk 5 besproken zijn, kunnen gaan vervangen. (Zie Fabry, 1974, voor een sterk pleidooi voor een dergelijke ontwikkeling.)

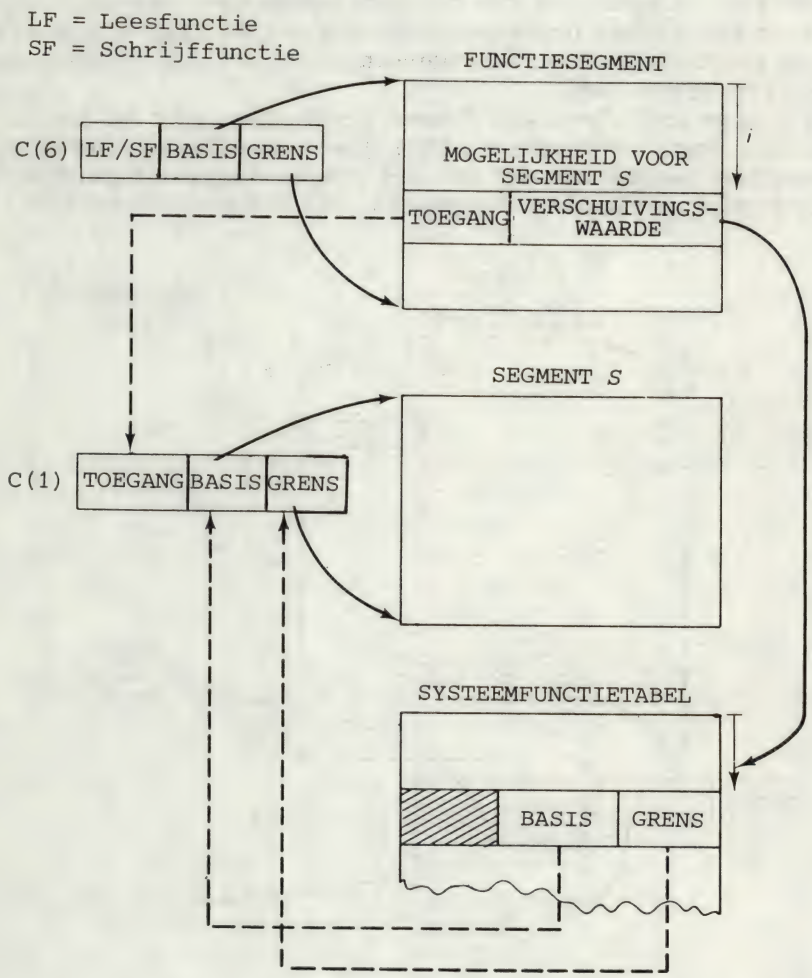
Een functie bestaat in de Plessey 250 uit de toegangsindicator, het basisadres en het grensadres van een bepaald segment. De functies van het huidige proces liggen opgeslagen in een stel processorregisters, die toegankelijk zijn voor het programma en die *functieregisters* genoemd worden, en de programma-adressen worden aangegeven door het geven van het nummer van een functieregister samen met een verschuivingswaarde. De gewenste geheugenplaats wordt verkregen door het toevoegen van de verschuivingswaarde aan het basisadres dat opgeslagen is in het functieregister. Het grensadres en de toegangsindicator worden gebruikt voor de controle op geheugen- en toegangsschendingen.

De lezer kan zich hier afvragen of de set registers voor de functies, afgezien van snelheidsverschillen, in principe niet hetzelfde is als de traditionele segmenttabel. Het onderscheid is dat de toewijzing van functieregisters onder besturing van plaatselijke programma's ligt en dat de programmeur of het vertaalprogramma vrij is de registers naar eigen believen anders toe te wijzen. In tegenstelling hiermee worden ingangen in segmenttabellen door het besturingssysteem op een algemene basis toegewezen, en als zij eenmaal toegewezen zijn veranderen zij niet meer van plaats.

Omdat het economisch gunstig is slechts te voorzien in een klein aantal registers, worden de functies van een proces ook opgeslagen in een functiesegment in het geheugen en al naar gelang dat vereist is in de registers geladen. Zodoende geeft het functiesegment dat op het huidige moment toegankelijk is voor een proces, zijn functies weer en definieert het domein waarin het moet lopen. De toegangsvoorrechten 'leesfunctie' en 'schrijffunctie' die wezenlijk anders

zijn dan het normale 'lees' en 'schrijf' en die alleen slaan op het functiesegment, worden gebruikt om te voorkomen dat gewone gegevens behandeld worden als functies. Dit voorkomt dat functies in gegevenssegmenten worden aangemaakt en daarna in functieregisters geladen worden. Volgens afspraak wordt register C(6) gebruikt voor de adressering van het functiesegment van het huidige proces en wordt register C(7) gebruikt voor de adressering van het segment dat de code bevat die uitgevoerd wordt.

Er is een verdere verfijning nodig voor het mogelijk maken van het gemeenschappelijk gebruik door processen van segmenten, zoals hierboven beschreven is. In het hier beschreven schema zou het verplaatsen van een gemeenschappelijk gebruikt segment in het

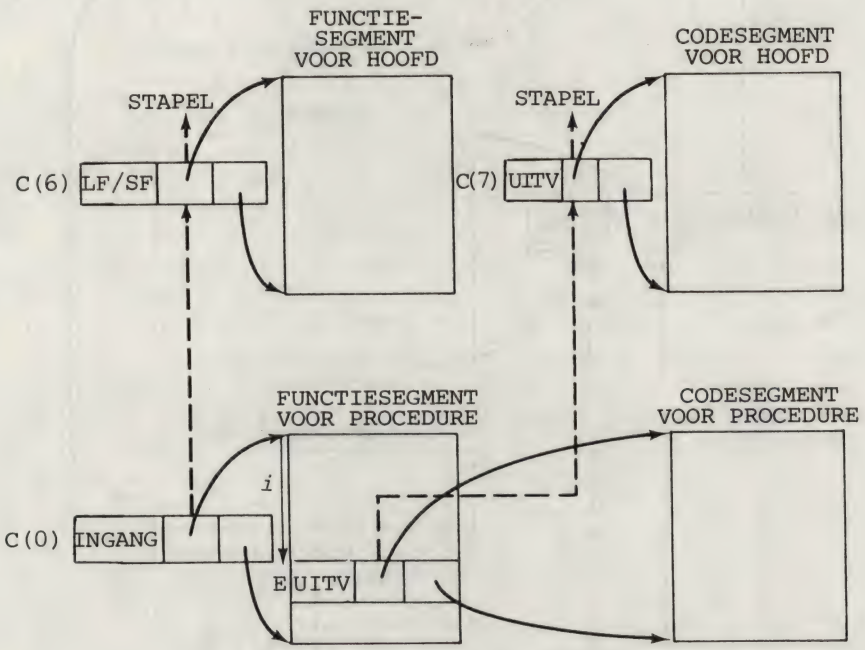


Figuur 9.5 *Handeling voor 'laad functie' (C(1) := C(6) + i)*

geheugen een verandering met zich meebrengen van de overeenkomstige functies in de functiesegmenten van alle betrokken processen. Om dit te voorkomen worden de basis- en grensadressen van alle segmenten in een *systeemfunctietabel* gehouden en alle opgeslagen functies verwijzen via deze tabel individueel naar segmenten. Het verplaatsen van een segment brengt zodoende alleen een verandering van de systeemfunctietabel met zich mee en niet van de ingangen in de individuele functiesegmenten. Figuur 9.5 geeft het laden van het functieregister C(1) weer, met een mogelijkheid voor segment *S* dat in de *i*-de ingang van het functiesegment staat.

Er moet opgemerkt worden dat de functietabel van het systeem zelf een segment is dat geadresseerd wordt door een speciaal functieregister en dat slechts voor bepaalde delen van het besturingssysteem, die zich bezighouden met het geheugenbeheer, toegankelijk is. Het is voor alle andere processen onzichtbaar, zij kunnen de functies in de functiesegmenten beschouwen als direct verwijzend naar de betrokken segmenten.

Een proces kan alleen van domein veranderen (dat wil zeggen zijn functiesegment veranderen) als het een speciale 'ingangsfunctie' heeft voor het functiesegment van het nieuwe domein. Figuur 9.6 geeft de manier aan waarop een proces, dat het programma HOOFD



Figuur 9.6 *Verandering van het domein waarin de uitvoering plaatsvindt (CALL C(0) + i; call = aanroepen)*

uitvoert, een procedure SUB in een ander domein aanroept. In het begin adresseert C(6) het functiesegment voor HOOFD; C(7) bevat een 'uitvoerfunctie' voor het segment met de programmacode voor HOOFD. Het proces bezit in C(0) een ingangsfunctie voor het functiesegment van SUB, dat op zijn beurt in zijn *i*-de plaats een uitvoerfunctie E bevat voor het codesegment van SUB. Het aanroepen van een procedure waarbij C(0) wordt aangegeven samen met een verschuivingswaarde *i*, heeft tot gevolg dat C(6) vervangen wordt door de ingangsfunctie in C(0) en dat C(7) vervangen wordt door uitvoerfunctie E. Daardoor verwijzen C(6) en C(7) nu naar de juiste functie- en codesegmenten in het nieuwe domein. Een voorrecht voor een leesfunctie is aan C(6) toegevoegd om de aangeroepen procedure in staat te stellen zijn eigen functies te lezen, en de vorige waarden van C(6) en C(7) worden op een stapel gelegd om weer teruggehaald te worden bij de erop volgende terugkeer. Noch de aanroeper noch de aangeroepen procedure hebben toegang tot elkaars functies, uitgezonderd die welke als parameters doorgegeven mogen worden in registers C(1) tot en met C(5). Het schema levert daarom een volledige bescherming op bij elkaar wantrouwende procedures.

9.5 TEN SLOTTE

We hebben in dit hoofdstuk gezien dat de behoefte aan bescherming een logisch gevolg is van een situatie waarin meerdere gebruikers toegang hebben tot een systeem. Wanneer er steeds meer grootschalige gegevensbestanden en een groeiende zorg aangaande de privacy optreden, worden veiligheid en betrouwbaarheid steeds urgenter. Alhoewel er van oudsher in meer of mindere mate in bescherming werd voorzien door een diversiteit aan mechanismen op diverse punten in het systeem, wordt het nu mogelijk een geïntegreerde benadering te verzorgen. Uitgaande van geschikte hardware zouden we in ons papieren besturingssysteem een beschermingsschema willen inbouwen dat vergelijkbaar is met een van de schema's die hierboven beschreven zijn.

10 Betrouwbaarheid

In sectie 2.3 hebben we gesteld dat betrouwbaarheid een van de zeer gewenste eigenschappen van een besturingssysteem is; een onbetrouwbaar systeem is zelfs van erg weinig nut. In dit hoofdstuk bekijken we nader wat er met betrouwbaarheid bedoeld wordt en bespreken we manieren voor het verbeteren van de betrouwbaarheid van een besturingssysteem. We zullen zien dat betrouwbaarheid geen accessoire is dat zomaar toegevoegd kan worden, maar een vereiste waarmee vanaf de eerste stadia van het systeemontwerp rekening gehouden moet worden.

10.1 DOELSTELLINGEN EN TERMINOLOGIE

Uit de voorgaande hoofdstukken zal duidelijk zijn dat een besturingssysteem een complex stuk software is waarvan verwacht wordt dat het een diversiteit aan functies voor een verscheidenheid van gebruikers uitvoert. Het is voor de gebruikers belangrijk dat deze functies juist uitgevoerd worden en een van de eerste doelstellingen van de systeemontwerper is de zorg hiervoor. De taak van de ontwerper wordt er niet gemakkelijker op, doordat het voltooide product niet in een perfecte wereld gaat werken; het zal waarschijnlijk onderworpen zijn aan een grote verscheidenheid van omstandigheden die zijn werking mogelijk negatief kunnen beïnvloeden. Dergelijke omstandigheden zijn onder andere het niet goed functioneren van de hardware, procedurefouten van de mensen die de computer bedienen en foutieve of zinloze verzoeken die door de gebruiker gedaan worden. Zelfs in dergelijke ongunstige omstandigheden zou een besturingssysteem, mogelijk op een verlaagd prestatieniveau, dienst moeten blijven doen.

Deze opmerkingen leiden ons tot de volgende definitie van de *betrouwbaarheid* van een besturingssysteem: het is de mate waarin het voldoet aan de specificaties betreffende de dienstverlening van gebruikers, zelfs als het onderhevig is aan onverwachte en vijandige omstandigheden. Deze definitie benadrukt het feit dat betrouwbaarheid eerder een relatief dan een absoluut begrip is: geen enkel systeem kan totaal betrouwbaar zijn; het kan bijvoorbeeld onmogelijk functioneren in het geval van een gelijktijdige uitval van alle

hardware-onderdelen van de computer. Een zeer betrouwbaar besturingssysteem zal, zelfs onder grote druk van hardwarestoringen of fouten van gebruikers, aan zijn specificatie blijven voldoen. Een minder betrouwbaar systeem kan al door een zinloos verzoek afwijken van zijn specificaties.

Het betrouwbaarheidsniveau dat gehaald moet worden door een besturingssysteem hangt natuurlijk af van het vertrouwen dat de gebruikers erin stellen. Een groot vertrouwen, zoals bij een systeem dat een ruimtevaartuig bestuurt of dat de toestand van patiënten in een ziekenhuis bewaakt, vereist een grote betrouwbaarheid; is het vertrouwen wat minder, zoals bij een systeem voor het bewerken of terughalen van documenten, dan is minder betrouwbaarheid vereist. Omdat een hoge betrouwbaarheid meestal hoge kosten met zich meebrengt (zoals we later zullen zien), moet de ontwerper op een betrouwbaarheidsniveau mikken dat overeenkomstig het door de gebruikers in het systeem gestelde vertrouwen is. (Het is echter interessant op te merken dat de betrouwbaarheid van het systeem ook van invloed is op het door de gebruikers erin gestelde vertrouwen.)

Het begrip betrouwbaarheid moet niet verward worden met *correctheid*: de begrippen zijn met elkaar verbonden maar verschillend. Een besturingssysteem is *correct* indien het in een bepaalde omgeving een bepaald (gewenst) gedrag ten toon spreidt. Correctheid is zeker een gewenste bijkomstigheid van een besturingssysteem, maar het is een onvoldoende voorwaarde voor betrouwbaarheid. Dit komt omdat het aantonen van correctheid, of dat nu door testen of door formeel bewijzen verkregen wordt, uitgaat van veronderstellingen over de werksituatie die in het algemeen niet opgaan. Deze betreffen meestal de aard van de invoer en het correct werken van de hardware en kunnen in de praktijk volledig ongegrond blijken. Met andere woorden, de werksituatie van een besturingssysteem komt zelden overeen met die waarvan bij het aantonen van correctheid werd uitgegaan.

Correctheid is niet alleen een onvoldoende voorwaarde voor betrouwbaarheid, maar, en dat is misschien verrassender, is ook niet noodzakelijk. Terwijl delen van het besturingssysteem niet correct kunnen zijn, in die zin dat de algoritmen die een bepaald proces besturen niet het vereiste resultaat geven, kan het systeem toch nog betrouwbaar werken. Laten we, om een voorbeeld te geven, eens een opslagsysteem bekijken waarvan de *sluit*routine in bepaalde omstandigheden vergeet de lengte van een bestand in de bijbehorende indexingang vast te leggen. Op voorwaarde dat het bestand eindigt met een herkenbaar 'einde van bestand' teken, kunnen de I/O routines het bestand nog steeds met goed gevolg lezen en zal het systeem betrouwbaar werken, zelfs als in aanmerking genomen wordt dat een van zijn onderdelen niet goed werkt. Dit voorbeeld laat het gebruik van overbodige of overtollige informatie zien voor het camoufleren en herstellen van fouten; dit is een techniek waarop we later zullen terugkomen.

De lezer mag hieruit niet afleiden dat de correctheid onbelangrijk is. Het kan een voorwaarde zijn die noch voldoende noch vereist is voor de betrouwbaarheid, maar het helpt zeker. Een besturingssysteem dat correct is zal ongetwijfeld betrouwbaarder zijn dan een besturingssysteem dat dat niet is. Het zou zeker een slechte gewoonte van een ontwerper zijn als hij op de betrouwbaarheid van mechanismen zou vertrouwen voor het verbergen van bekende gebreken, of dat zelfs maar zou gebruiken voor de rechtvaardiging van het feit dat hij zich niet maximaal inspant voor het bereiken van de correctheid. De juiste gang van zaken is trachten een systeem te maken dat correct werkt ten aanzien van gespecificeerde aannames over de werkomgeving en tegelijkertijd betrouwbaarheidsmechanismen te ontwerpen voor het opvangen van die gevallen waarbij de aannames ongeldig zijn. We zullen in volgende secties deze aanvullende benaderingen bekijken.

We introduceren hier nog drie termen. We definiëren een *fout* als een afwijking door het systeem van zijn opgegeven gedrag. Een fout is dus een gebeurtenis; voorbeelden zijn het toewijzen van een ondeelbare hulpbron aan twee processen, of het wissen van een indexingang voor een bestand dat nog gebruikt wordt. Een fout kan veroorzaakt worden door een storing in de hardware, door een onverwachte handeling van degene die de computer gebruikt of bedient, of door een onvolkomenheid (Engels: 'bug') in een van de programma's in het systeem. In al die gevallen gebruiken we de term *storing* voor het aangeven van de oorzaak van een fout. Als een fout plaatsvindt is het waarschijnlijk dat er delen van de informatie in het systeem verminkt worden. We verwijzen met het woord *schade* naar de verminkingen die door een fout zijn veroorzaakt; schade kan natuurlijk tot storingen leiden die weer verdere fouten tot gevolg hebben. We zullen zien dat één van de manieren voor het bereiken van een hoge betrouwbaarheid het beperken van de schade ten gevolge van een fout is; daardoor wordt de voortwoekering van fouten door het systeem beperkt.

Uit de voorgaande bespreking kunnen we afleiden dat pogingen voor het maken van een zeer betrouwbaar besturingssysteem toegespitst moeten worden op de volgende gebieden:

(a) *storingsvermijding*: het tijdens de ontwikkeling en implementatie elimineren van storingen; dat wil zeggen het maken van een correct systeem;
(b) *foutherkenning*: het zo snel mogelijk herkennen van fouten voor het verminderen van de veroorzaakte schade;
(c) *storingsbehandeling*: het identificeren en uitschakelen van iedere storing die een fout tot gevolg heeft;
(d) *foutcorrectie*: het vaststellen en herstellen van de schade die het gevolg is van een fout.

In de volgende secties zullen we deze gebieden één voor één behandelen.

10.2 HET VERMIJDEN VAN STORINGEN

Zoals we al gezien hebben, kunnen storingen in een computer veroorzaakt worden door de hardware, door de incompetentie en onwetendheid van de gebruiker of bediener van de computer, of door 'bugs' in de software. Hoe elk van deze soorten fouten vermeden kan worden, wordt hieronder besproken.

(1) Storingen veroorzaakt door gebruiker en bediener

Het enige dat we zullen zeggen over storingen die veroorzaakt zijn door gebruiker of bediener is, dat zij nooit uitgesloten kunnen worden: we kunnen alleen maar hopen dat ze in aantal afnemen door het gebruik van goede oefen- en leerprogramma's voor de gebruikers.

(2) Storingen veroorzaakt door de hardware

De meest voor de hand liggende manier voor het verminderen van hardwarestoringen is het toepassen van de betrouwbaarste onderdelen die, met inachtneming van de kosten, verkregen kunnen worden. De ontwerpers van de hardware gebruiken diverse methoden voor het verhogen van de betrouwbaarheid, vanaf de meest voor de hand liggende - het gebruik van individueel betrouwbare onderdelen - tot complexere technieken, die een vorm van foutherkenning en foutcorrectie in ieder subsysteem toepassen. De foutherkenning is meestal gebaseerd op het vastleggen of doorseinen van overbodige informatie, zoals pariteitsbits en controletotalen; door de handeling die de fout veroorzaakt te herhalen, probeert men de fout te corrigeren. Een voorbeeld voor deze techniek is de magnetische band- en schijfapparatuur, die zo ontworpen is dat handelingen een beperkt aantal keren herhaald worden als zich een fout voordoet in de pariteit of in het controletotaal. Het opnieuw proberen heeft natuurlijk alleen zin als het een tijdelijke storing betreft (dat wil zeggen storingen die het gevolg zijn van tijdelijke omstandigheden, zoals interferentie of stofdeeltjes); blijvende storingen zullen in fouten blijven uitmonden.

Een andere techniek die bij de gegevensoverdracht veel toegepast wordt, is het gebruik van foutherkennende en -corrigerende codes waarbij de overtollige informatie, die samen met de gegevens wordt verzonden, gebruikt wordt als middel voor het herstellen van bepaalde fouten die bij het verzenden zijn opgetreden. Nog een andere techniek is die waarbij *de meeste stemmen gelden*. Hierbij krijgen meerdere (meestal drie) onderdelen dezelfde invoer waarna hun uitvoer wordt vergeleken. Als de uitvoer verschillend is, dan wordt aangenomen dat het meerderheidsoordeel het juiste is en het onderdeel, dat het minderheidsoordeel had, wordt als verdacht

gebrandmerkt. Van een onderdeel dat herhaaldelijk in de minderheid is, wordt aangenomen dat het defect is en vervangen kan worden. Deze meerderheidstechniek voor de betrouwbaarheid is kostbaar omdat ze gebaseerd is op het kopiëren van onderdelen en ze wordt alleen gebruikt als er echt hoge eisen aan de betrouwbaarheid gesteld worden.

Het doel van al deze technieken is het *maskeren* van de hardwarestoringen voor de software die erop loopt. Dat wil zeggen dat er storingen in de hardware kunnen voorkomen, maar dat de gevolgen daarvan voor de software verborgen blijven, zodat wat betreft de software de hardware storingsvrij lijkt. We zullen in sectie 10.6 zien dat het begrip maskering uitgebreid kan worden tot binnenin het besturingssysteem zelf, zodat fouten die binnen een niveau optreden verborgen blijven voor de erbuiten liggende niveaus.

(3) Storingen in de software

Er is een aantal, elkaar aanvullende, benaderingen voor het terugdringen van de aanwezigheid van fouten in de software. Drie van de belangrijkste bepreken we hieronder.

De eerste benadering is het toepassen van beheersings- en werkmethoden voor het programmeren van softwarehulpmiddelen die een positieve bijdrage vormen voor het produceren van een storingsvrij produkt. In het midden van de jaren '60 dacht men in bepaalde groepen dat er aan het ontwerpen van een groter stuk software ook meer programmeurs gezet moesten worden. De ervaring van IBM, die een compleet leger programmeurs aan de ontwikkeling van OS/360 zette, verwees deze mythe naar het rijk der fabelen en men erkent nu dat het ongebreideld toepassen van mankracht waarschijnlijk meer problemen veroorzaakt dan het oplost. Het samenvoegen van programmeurs in kleine groepen, algemeen bekend als 'programmeringsteams', wordt tegenwoordig als een betere manier voor het organiseren van de softwareproduktie beschouwd. Iedere groep is verantwoordelijk voor een softwaremodule die exact gedefinieerde externe specificaties en koppelingen heeft. Binnen de groep heeft ieder lid een nauw omschreven functie, zoals het coderen, de documentatie of de communicatie met andere groepen. Een gedetailleerde beschouwing van dergelijke organisatorische gewoonten ligt buiten ons huidige bereik: Brooks (1975) geeft een verhelderende en onderhoudende beschrijving van de diverse valstrikken.

De manier waarop programmeurs te werk gaan bij het schrijven van programma's kan een aanzienlijke invloed hebben op de kwaliteit van het voltooide produkt wat betreft de samenhang, begrijpelijkheid en storingsvrijheid. Er zijn de afgelopen jaren diverse werkmethoden gepropageerd, waarvan het *gestructureerde programmeren* (Dahl e.a., 1972; Wirth, 1971) en zijn varianten (bijvoorbeeld Yourdon, 1975; Jackson, 1975) de grootste bekendheid genieten. Het gestructureerde programmeren heeft elders genoeg aandacht gekregen, zodat we er hier niet verder op ingaan: de lezer die er

meer over te weten wil komen verwijzen wij naar de bovenstaande titels of naar Alagic, Arbib (1978) en Ledgard.

De beste softwarehulpmiddelen voor het maken van een storingsvrij produkt zijn waarschijnlijk een editor, een macro-assembler en een vertaalprogramma voor een hogere programmeertaal. We hebben in sectie 4.2 al opgemerkt dat de enige delen van een besturingssysteem, die in machinegerichte taal geschreven moeten worden, die zijn welke in directe verbinding staan met de hardware van de machine, te weten delen van de kern en de besturing van de I/O apparatuur. Het gebruik van een hogere programmeertaal voor de rest van het systeem versnelt de produktie van bruikbare code en maakt de uitsluiting van storing mogelijk. Er wordt soms gezegd dat hogere programmeertalen ongeschikt zijn voor het schrijven van besturingssystemen, omdat de aangemaakte werkcode bij de uitvoering minder efficiënt is dan de overeenkomstige met de hand geschreven code. Er zijn ten minste drie zaken die dit beeld tegenspreken: ten eerste, een goed optimaliserend vertaalprogramma kan vaak een betere code produceren dan een goede machinetaalprogrammeur; ten tweede, die stukken inefficiëntie die blijven, kunnen - indien nodig - met de hand geoptimaliseerd worden; en ten derde is betrouwbaarheid een doel dat minstens even belangrijk is als efficiëntie.

De tweede algemene benadering voor het verminderen van de frequentie waarin softwarestoringen voorkomen, is het met min of meer formele middelen proberen te bewijzen dat alle programma's en hun wisselwerkingen in het besturingssysteem correct zijn. Dit is een gigantische opdracht die voor de grote systemen buiten het bereik van de huidige mogelijkheden ligt. Pionierswerk van Floyd (1967), Hoare (1972) en Dijkstra (1976) heeft echter de mogelijkheid geopend dat het in de nabije toekomst mogelijk zal zijn dergelijke bewijzen te leveren. Zelfs nu is het al mogelijk een behoorlijk vertrouwen in de correctheid van een systeem te krijgen door het bewijs dat een aantal van de programma's, die er deel van uitmaken, correct zijn.

De bekendste benadering voor het bewijzen dat een programma correct is (Alagic en Arbib, 1978; Hantler en King, 1976), is het formuleren van stellingen (meestal in de vorm van beweringsberekeningen) die waar moeten zijn tijdens diverse stappen van de uitvoering van het programma. Vanuit de aanname van de eerste stelling gaat de bewijsvoering verder met het onderzoek of de uitvoering van de programma-opdrachten inderdaad de volgende stellingen waar maakt. De belangrijkste moeilijkheden die men tegenkomt zijn het formuleren van de juiste stellingen en het vrij bewerkelijke stap voor stap bewijzen dat zij waar zijn.

De derde benadering is het elimineren van softwarestoringen door middel van het systematisch testen. Het testen is altijd een van de belangrijkste methoden geweest voor het herkennen van storingen, maar er is verrassend weinig overeenstemming over wat de juiste werkwijze is. In het ideale geval zou men een programma willen testen met alle mogelijke invoer, maar dat is natuurlijk onuitvoerbaar; het enige waarop men kan hopen is, dat de gekozen

testgegevens alle overgebleven storingen aan het licht zullen brengen. Er bestaat geen algemene manier voor het kiezen van de gegevens waarmee dit gedaan moet worden, hoewel er wel een aantal pragmatische richtlijnen is voorgesteld. Een van die richtlijnen is het kiezen van gegevens die alle mogelijke wegen door het programma zullen betreden. Een andere is het kiezen van gegevens die aan de grenzen van het toelaatbare bereik liggen: zo is nul bijvoorbeeld een voor de hand liggende keuze in alle situaties waarbij de invoer een positief geheel getal moet zijn.

Als er een groot aantal programma's uitgevoerd moet worden, cumuleren de moeilijkheden bij het testen door een aantal elkaar beïnvloedende processen, zoals dat bij een besturingssysteem het geval is. Niet alleen dat elk programma getest moet worden, maar ook alle koppelingen tussen de verschillende programma's en processen. Alleen als men de koppelingen zo eenvoudig mogelijk houdt, kan men hopen dat dit alles met goed gevolg gedaan kan worden.

We kunnen deze bespreking samenvatten met de opmerking dat op geen van de bovenstaande benaderingen - de werkwijze voor het programmeren, formele correctheidsbewijzen en het testen - vertrouwd kan worden voor het verkrijgen van een storingsvrij produkt. De technieken worden nog verder ontwikkeld, maar het is redelijk te stellen dat een combinatie van de drie benaderingen zelfs nu al het vertrouwen in de correctheid van de geproduceerde software aanzienlijk kan doen toenemen.

10.3 FOUTHERKENNING

De sleutel tot het herkennen van fouten is *overvloedigheid* (Engels: *redundancy*), dat wil zeggen het geven van 'overbodige' informatie die voor de controle op de geldigheid van de 'hoofdinformatie' in het systeem gebruikt kan worden. De term 'overvloedigheid' geeft weer dat de informatie, die voor de controle gebruikt wordt, overvloedig is voor zover het de belangrijkste algoritmen van het systeem aangaat. We hebben in de vorige sectie gezien hoe overvloedige informatie, zoals pariteitsbits en controletotalen, gebruikt kunnen worden voor foutherkenning. Het coderen is ook een nuttig middel voor de foutherkenning en kan in sommige gevallen ook gebruikt worden voor het herstellen van de fout. Zoals eerder opgemerkt, kunnen fouten die door de hardware herkend worden door de hardware zelf gemaskeerd worden, of ze kunnen met behulp van foutvallen in de basis-niveau ingreep besturing (BNIB) aan het besturingssysteem gemeld worden (sectie 4.4). Voorbeelden van deze laatste soort fouten zijn rekenkundige overloop, geheugen- en beschermingsschendingen. De actie die door het besturingssysteem in dergelijke gevallen ondernomen kan worden, wordt verderop in dit hoofdstuk besproken.

Foutherkenning kan ook door het besturingssysteem zelf uitgevoerd worden. Een veelgebruikte controlevorm voor processen is de controle op de samenhang in de gegevensstructuren waar ze gebruik van maken. Een voorbeeld hiervan zou voor de processorwachtrij kunnen zijn dat deze de vorm heeft van een dubbel gekoppelde lijst waarbij de werkindeler de koppelingen zowel voorwaarts als achterwaarts opspoort, onafhankelijk van waar hij de wachtrij ingaat. De overvloedige informatie die in dit geval verschaft wordt zijn de achterwaartse koppelingen: alleen de voorwaartse zijn nodig voor de belangrijkste algoritmen voor de werkindeler en het verdeelprogramma. Een ander voorbeeld is de controle op veranderingen in tabellen door het in de tabel bewaren van een controletotaal van al zijn ingangen. Als er een ingang veranderd wordt, kan het gevolg op het controletotaal met behulp van een rekensom voorspeld worden. Als het controletotaal na de verandering niet overeenkomt met het voorspelde totaal, dan is er een fout opgetreden tijdens de wijziging.

Een algemenere vorm van deze controle is het koppelen van een *acceptatietest* aan iedere belangrijke handeling van een proces; die kan dan als maatstaf gebruikt worden om te bekijken of de handeling naar behoren is uitgevoerd. De acceptatietest is een Booleaanse functie die bekeken wordt (dit is een deel van de uitvoering van het proces) nadat de handeling is voltooid. Als het resultaat 'waar' is, wordt aangenomen dat de handeling correct is uitgevoerd; als het resultaat 'onwaar' is, heeft er zich een fout voorgedaan. De acceptatietest kan zo strikt zijn als nodig gevonden wordt; hij zal zo opgesteld worden dat er op die fouten gecontroleerd wordt waarvan de ontwerper denkt dat ze het meest frequent optreden. Bekijk als voorbeeld eens de handelingen van de werkindeler als die regelmatig de prioriteiten van de processen verandert en de processorwachtrij opnieuw ordent. De acceptatietest voor deze handeling kan simpelweg de controle zijn of de procesbeschrijvers in de nieuwe wachtrij inderdaad in volgorde van prioriteit staan. Een striktere test heeft als bijkomende voorwaarde dat het aantal beschrijvers in de wachtrij hetzelfde is als voordien; er wordt zo voor gewaakt dat er geen beschrijver onbedoeld verdwenen of gedupliceerd is. Er wordt natuurlijk meestal een compromis gesloten tussen de striktheid van de acceptatietest en het extra werk dat het uitvoeren ervan met zich meebrengt, een compromis dat bij alle foutherkenningsmechanismen gesloten moet worden. De ontwerper moet de nadelen, kostbare apparatuur of een vermindering van de prestaties, afwegen tegen de voordelen die eruit voortvloeien. Het is helaas vaak zo dat noch de kosten, noch de baten precies gemeten kunnen worden.

Tot nu toe hebben we alleen fouten behandeld die binnen een enkel hardware- of software-onderdeel optreden. Fouten die optreden bij koppelingen tussen de onderdelen zijn moeilijker te herkennen, maar het nagaan van de geloofwaardigheid van alle informatie die door de koppeling komt kan wel wat opleveren. Er kan bijvoorbeeld van een aantal procedureparameters gecontroleerd worden of zij binnen een juist bereik vallen en men kan mogelijk controleren of mededelingen die tussen processen overgedragen worden aan het een of andere vastgestelde protocol voldoen.

We zullen deze sectie besluiten met de opmerking dat herkenning van een fout in een vroeg stadium de beste manier is voor het beperken van de schade die erdoor wordt aangericht en van de verspilling. De mogelijkheid voor een besturingssysteem om na het optreden van fouten snel te reageren kan aanzienlijk verbeterd worden door het voorzien in geschikte hardware-beschermingsmechanismen. Daardoor kunnen de beschermingsmechanismen, die in hoofdstuk 9 zijn besproken, niet alleen een belangrijke bijdrage leveren aan de veiligheid van een besturingssysteem, maar ook aan de betrouwbaarheid ervan. (Zie Denning, 1976, voor een verdere uitwerking hiervan.)

10.4 HET VERHELPEN VAN STORINGEN

Het verhelpen van een storing houdt in dat eerst zijn plaats bepaald wordt, waarna hij hersteld kan worden. Zoals iedere systeemprogrammeur zal kunnen vertellen, houdt het herkennen van een fout en het opsporen van de bijbehorende storing bepaald niet hetzelfde in. Een fout kan diverse mogelijke oorzaken hebben die òf in de hardware òf in de software zitten en geen enkele hoeft voor de hand te liggen. Een van de belangrijkste hulpmiddelen bij het identificeren van een storing is het vaststellen van fouten vóórdat hun oorzaak verduisterd wordt door gevolgschade en de erop volgende fouten. Daarom is de herkenning van fouten in een vroeg stadium, zoals aan het einde van sectie 10.3 opgemerkt is, van het grootste belang.

Soms is het mogelijk de weg van de minste weerstand te kiezen door de storing volledig te negeren, maar dat houdt aannamen in over de regelmaat waarin de storing voorkomt en over de grootte van de schade die aangericht kan worden. Men kan het bijvoorbeeld niet de moeite waard vinden de plaats op te zoeken van een soms optredende hardwarestoring, totdat de regelmaat waarmee dit fouten veroorzaakt over een bepaalde drempel heengaat, zodat dit niet acceptabel meer is (Avizienis, 1977). Meestal is het echter belangrijk dat de storing gevonden en opgelost wordt. Het zoeken naar een storing zal meestal in goede banen geleid worden door het begrip dat de onderzoeker heeft van de systeemstructuur. Als de systeemstructuur onvolledig is of als de storing deze aangetast heeft, zal de taak van de onderzoeker behoorlijk moeilijk zijn.

Een manier om de onderzoeker te helpen is het systeem een spoor of logboek te laten maken van zijn laatste handelingen. Gebeurtenissen die in het logboek zijn vastgelegd, zouden de volgende kunnen zijn: het in werking stellen van processen, het binnengaan van procedures, I/O overdrachten, enzovoort. Helaas moet de hoeveelheid en detaillering van de informatie die vastgelegd wordt erg groot zijn, wil het logboek enige praktische waarde hebben, en het extra werk, nodig voor het vastleggen, kan erg veel zijn. Mogelijk levert het optioneel maken van het vastleggen van het spoor wat

op, zodat dit alleen maar ingeschakeld hoeft te worden als er redenen zijn om aan te nemen dat het systeem niet goed werkt. Ook het selectief maken van het spoor is zinnig, zodat alleen de verdachte delen van het systeem hun activiteiten hoeven vast te leggen. Dit vereist natuurlijk een subjectief oordeel - zelf ook voor fouten vatbaar! - over welke delen verdacht zijn.

Als een storing eenmaal gevonden is, is er voor het verhelpen een reparatie nodig. In het geval van een hardwarestoring houdt de reparatie gewoonlijk de vervanging van een defect onderdeel in. Dit kan met de hand of automatisch gedaan worden, afhankelijk van het feit of de hardware in staat is de plaats van zijn eigen storingen te bepalen en of de hardware zelf het defecte onderdeel kan 'uitschakelen'. Het met de hand vervangen kan tot gevolg hebben dat het hele systeem tijdelijk zijn diensten moet staken, of parallel - mogelijk op een minder hoog niveau - kan doorgaan met het verlenen van die diensten. Dit tweede heeft duidelijk de voorkeur. De reparatie van een schijfstation zou bijvoorbeeld de normale dienstverlening niet mogen onderbreken indien er andere stations beschikbaar zijn, maar de vervanging van een gedrukte schakeling in de centrale verwerkingseenheid van een systeem dat maar één enkele CVE heeft zal waarschijnlijk het volledig afsluiten en herstarten van het systeem noodzakelijk maken.

Storingen in de software-onderdelen komen voort uit een gebrekkig ontwerp en implementatie, of uit verminkingen die het gevolg zijn van vorige fouten. (In tegenstelling tot hardware heeft software geen last van storingen door veroudering.) Het verwijderen van een ontwerp- of implementatiestoring houdt meestal de vervanging in van een (hopelijk klein) aantal programmaregels, terwijl een verminkt programma vervangen kan worden door een reservekopie die elders is opgeslagen. Het repareren van verminkte gegevens is een onderwerp dat we voor de volgende sectie bewaren, die over het herstellen van fouten in het algemeen gaat. Net zo als met hardware zou men hopen dat software-onderdelen vervangen kunnen worden zonder dat het nodig is het hele systeem plat te leggen. Het systeem zou dan ook vanuit dit uitgangspunt ontworpen moeten worden.

10.5 HET HERSTELLEN VAN FOUTEN

Het herstellen van een fout houdt in dat er vastgesteld moet worden wat de schade is, gevolgd door een poging die te repareren. Het vaststellen van de schade kan helemaal gebaseerd zijn op een 'a priori' redenering van de onderzoeker. Ook is het mogelijk dat het systeem zelf gebruikt wordt voor het uitvoeren van een aantal controles om de schade die zich heeft voorgedaan vast te stellen. In beide gevallen zal het vaststellen (zoals bij de identificatie van storingen) plaats vinden aan de hand van een verondersteld oorzakelijk

verband dat door de systeemarchitectuur aangegeven wordt. Van een fout in het bijwerken van een bestandsindex mag bijvoorbeeld verwacht worden dat die het opslagsysteem beschadigt, maar niet de processtructuur van de apparaatbeschrijvers. Er bestaat natuurlijk gevaar dat de systeemstructuur zelf beschadigd is en daarmee de veronderstelde verbanden, maar de waarschijnlijkheid dat dat gebeurt wordt aanzienlijk beperkt door het gebruik van de juiste hardware-beschermingsmechanismen (zoals in sectie 10.3 werd voorgesteld).

De normale benadering voor het repareren van de schade is het maken van kopieën van de beschadigde programma's in de toestand waarin ze verkeerden op het moment vóór de storing optrad. Deze benadering gaat uit van het bestaan van *herstelpunten* (of *controlepunten*) waarin voldoende informatie is opgeslagen over de toestand van het proces om dat later weer te kunnen herstellen, als dat nodig is. De vereiste informatie bestaat ten minste uit de vluchtige omgeving en een kopie van de procesbeschrijver; verder kan het ook een kopie omvatten van de inhoud van de door het proces gebruikte geheugenruimte. Het interval tussen de herstelpunten bepaalt de hoeveelheid verwerking die waarschijnlijk verloren gaat als er zich een fout voordoet. Het informatieverlies kan worden beperkt door *verificatiespoor* (Engels: *audit trail*) technieken, waarbij alle veranderingen in de toestand van het proces worden vastgelegd zoals ze plaatsvinden. Het herstellen van de fout bestaat dan uit het teruggaan tot aan het laatste herstelpunt en het vervolgens maken van de veranderingen in de toestand, die door het verificatielogboek worden aangegeven. De verificatiespoor technieken zijn vergelijkbaar met de gedeeltelijke dump van een opslagsysteem, zoals in sectie 7.5 beschreven werd, in feite kan men de gedeeltelijke dump beschouwen als een bijzondere vorm van een verificatiespoormechanisme.

Herstelpunten en verificatiesporen brengen het vastleggen van alle toestandsinformatie en de veranderingen daarop met zich mee. Een belangrijk alternatief is het *herstelblok*-schema (Randell, 1975), waarbij de enige toestandsinformatie die vastgelegd wordt, die is welke werkelijk veranderd is. Omdat het schema ook elementen van foutherkenning bevat, is het een korte omschrijving waard.

Een herstelblok is een programmadeel met de volgende structuur:

```
verzeker    acceptatietest door
            eerste alternatief
anders
            tweede alternatief
anders
            ander alternatief
anders          .
   .            .
   .            .
   .
```

Het eerste alternatief is het programmadeel dat normaliter wordt uitgevoerd; de andere alternatieven worden alleen gebruikt als het eerste alternatief niet werkt. Het al of niet werken van een alternatief wordt bepaald door de uitvoering van een acceptatietest (zie sectie 10.3) die bij het blok hoort. Zodoende houdt de uitvoering van het herstelblok de uitvoering van het eerste alternatief in, gevolgd door de uitvoering van de acceptatietest. Als de test slaagt, zoals gewoonlijk het geval zal zijn, wordt verondersteld dat het hele blok met succes is uitgevoerd. Als de test niet slaagt, wordt de toestand van het proces teruggebracht naar die, welke bestond voordat het eerste alternatief werd uitgevoerd. Daarna wordt het tweede alternatief uitgevoerd. De acceptatietest wordt weer uitgevoerd voor het bepalen van het eventuele succes en, als het nodig is, worden eventueel de volgende alternatieven opgeroepen. Als geen van de alternatieven werkt, dan wordt van het hele blok gezegd dat het niet werkt. Door het gebruik van een alternatief blok wordt dit euvel verholpen, als zo'n blok bestaat (alternatieve blokken kunnen tot elke diepte genest worden). Zo niet, dan wordt het proces herstart.

De alternatieven binnen een herstelblok kunnen als software-reserveonderdelen beschouwd worden, die automatisch gebruikt worden wanneer het eerste alternatief niet werkt. In tegenstelling tot hardware-reserveonderdelen hebben de alternatieven niet hetzelfde ontwerp en gebruiken ze meestal verschillende algoritmen. Het eerste alternatief gebruikt dát algoritme voor het vervullen van de functie van het blok, dat vanwege zijn efficiëntie of een andere reden het meest gewenst wordt; de andere alternatieven gebruiken minder efficiënte algoritmen of algoritmen die de gewenste functie op een nog acceptabele maar niet volledige manier vervullen. Het niet werken van een alternatief wordt als een zeldzame gebeurtenis beschouwd (vergelijkbaar met een fout ten gevolge van een éénmalige hardwarestoring) en het alternatief wordt alleen vervangen voor het blok dat op dit moment wordt uitgevoerd. Het is echter verstandig om alle fouten vast te leggen, zodat restfouten uit het ontwerp en het programmeren herkend en geëlimineerd kunnen worden.

Het herstelblok-schema gaat er bij zijn werking van uit dat het mogelijk is de toestand van een proces te herstellen als een acceptatietest niet slaagt. Het gemak waarmee dit gedaan kan worden, hangt ervan af of het betrokken proces al dan niet met andere processen heeft samengewerkt gedurende de uitvoering van het blok. Als dat niet het geval is, houdt het herstellen van zijn toestand slechts het herstellen van de oude waarden van de programmavariabelen en registers in, die veranderd zijn. Dit kan gedaan worden met een 'herstelreserve' (Horning e.a., 1974), dit is een deel van het geheugen waarin de waarden voor hun wijziging worden opgeslagen. Alleen die waarden die gewijzigd worden moeten in de reserve worden opgeslagen en waarden die meerdere malen gewijzigd moeten worden hoeven maar één keer opgeslagen te worden. Het niet werken van een alternatief zorgt ervoor dat de waarden uit de reserve worden

teruggezet voordat het volgende alternatief wordt geprobeerd; het slagen van een alternatief heeft het wissen van de waarden in de reserve tot gevolg, omdat de wijzigingen nu als juist beschouwd kunnen worden. Een nesting van de herstelblokken wordt verkregen door de reserves met behulp van de stapelmethode te organiseren.

Het geval, waarin interactie tussen processen heeft plaatsgevonden binnen een blok, is moeilijker te behandelen. Een benadering die mogelijk is, is alle interacties tussen processen te beschouwen als een 'gesprek' tussen die processen. Als er van een proces een reservekopie gemaakt moet worden tijdens het gesprek, dan moet dat ook van het andere proces gemaakt worden, omdat de informatie die overgezonden is tijdens het gesprek foutief geweest kan zijn. Daarom mag geen van de twee processen verder dan het einde van het gesprek, vóórdat ze beide geslaagd zijn voor de acceptatietest aan het einde ervan. Het niet slagen van een van de twee acceptatietesten heeft tot gevolg dat teruggegaan wordt naar de reservekopieën van de processen aan het begin van het gesprek. Het principe kan verder uitgewerkt worden voor gelijktijdige gesprekken tussen meerdere processen.

Deze beschrijving van het schema van de herstelblokken was noodgedwongen kort, maar de lezer zal inzien dat het diverse aspecten van storingstolerantie, die eerder besproken zijn, combineert. De foutherkenning wordt bereikt met behulp van acceptatietesten; de storingen worden verholpen door te voorzien in alternatieven; het herstellen is gebaseerd op het reservemechanisme. Een nadere beschrijving van het schema wordt in de aangehaalde referenties gegeven.

10.6 HET BEHANDELEN VAN FOUTEN OP MEERDERE NIVEAUS

In de vorige secties hebben we diverse technieken voor het behandelen van fouten beschreven. In deze sectie geven we een idee over hoe deze technieken opgenomen kunnen worden in de geslaagde structuur van ons papieren besturingssysteem.

Het belangrijkste doel is het voor ieder niveau in het besturingssysteem maskeren van fouten die zich in de daaronder liggende niveaus voordoen. Daarom moet ieder niveau in het systeem zoveel mogelijk verantwoordelijk zijn voor het herstellen van zijn eigen fouten, zodat het voor de bovenliggende niveaus vrij van fouten lijkt. Het idee is een uitbreiding binnen het besturingssysteem van de foutmaskering voor de hardware die in sectie 10.2 besproken is. In gevallen waarbij het maskeren onmogelijk is, moeten fouten die in een laag niveau voorkomen, wanneer dat voor een hoger niveau werkt, op een ordelijke manier aan dat hogere niveau gemeld worden (bijvoorbeeld door het terugsturen van een speciale foutmelding). Het hogere niveau kan dan zelf tot herstel in staat zijn, waardoor

de fout voor nog hogere niveaus gemaskeerd wordt. Die fouten, die helemaal niet gemaskeerd kunnen worden, moeten ten slotte natuurlijk aan de gebruiker of de beheerder gemeld worden. Op het laagste niveau van het systeem worden fouten, die herkend zijn door de CVE hardware maar die daardoor niet gemaskeerd kunnen worden, door middel van de foutvallen in de basis-niveau ingreep besturing gemeld aan de kern (sectie 4.4).

Laten we als voorbeeld van wat we bedoelen eens een gebruikersproces bekijken dat een bestand wil openen. Het proces betrekt er het opslagsysteem bij door het aanroepen van de *open*routine (sectie 7.6) die op zijn beurt weer het I/O systeem erbij betrekt door het aanroepen van de DOE IO procedure (sectie 6.2), die de gebruikersindex van de schijf afleest. De vereiste I/O opdracht wordt uiteindelijk gestart door de apparaatbesturing van de schijf. Als zich een pariteitsfout voordoet, kan in een eerste maskeringsniveau voorzien worden door de hardware van de besturingseenheid voor het schijfstation zelf. Deze is waarschijnlijk zo ontworpen dat het dergelijke fouten ontdekt en daarna het corresponderende blok opnieuw laat lezen. Als na een paar pogingen de situatie nog niet opgelost is, wordt de fout gemeld aan de apparaatbesturing door middel van het toestandsregister voor het apparaat dat door de besturingseenheid aan het einde van de overdracht opgesteld wordt. De apparaatbesturing kan proberen de fout te maskeren door het herstarten van de hele handeling, maar als dat niet werkt, moet de fout teruggemeld worden aan de *open*routine in het opslagsysteem (dat wil zeggen een foutmelding vanuit DOE IO). Binnen het opslagsysteem is het herstellen van de fout mogelijk als er nog ergens anders een kopie bestaat van de index: het I/O systeem kan verzocht worden de reservekopie te lezen. Als dat om de een of andere reden niet lukt, of als het opslagsysteem niet zo goed ontworpen is dat het reserve-indexen onderhoudt, dan kan de fout niet langer gemaskeerd worden en moet deze aan de gebruiker gemeld worden.

Een andere fout, die tijdens het openen van het bestand zou kunnen voorkomen, is de verminking van een wachtrijwijzer in de verzoekwachtrij van het schijfstation. Dit kan het gevolg zijn van een storing in zowel de hardware als in de programmering. Als de wachtrij georganiseerd is als een dubbel gekoppelde lijst, zoals in sectie 10.3 werd aangegeven, en als de routines die de wachtrij behandelen zowel voorwaartse als achterwaartse koppelingen opzoeken, dan kan het verloren gaan van het IOVB voor de overdracht van de index voorkomen worden. De schade aan de wijzer wordt gerepareerd door de behandelingsroutines voor de wachtrij zelf, waardoor de fout gemaskeerd wordt voor de apparaatbesturing en zodoende dus ook voor het opslagsysteem en het gebruikersproces.

In tegenstelling hiermee is het belangwekkend te zien wat er waarschijnlijk zal gebeuren als de verzoekwachtrij slechts één koppeling heeft, zodat de routines voor de behandeling van de wachtrij de fout niet kunnen ontdekken, laat staan maskeren. In dat geval zal het IOVB uit de wachtrij gehaald worden en zal de waarde van de seinpaal *verzoek wachtend* (sectie 6.2) niet overeenkomen met het

aantal verzoeken in de wachtrij. Er kunnen nu twee dingen gebeuren. De eerste mogelijkheid is dat de apparaatbesturing zal aannemen dat een geslaagde *passeer*handeling op *verzoek wachtend* betekent dat er een IOVB in de wachtrij aanwezig is. Hij zal dan een niet bestaand IOVB uit de wachtrij halen, zal de eruit voortkomende rotzooi beschouwen als een geldig I/O verzoek en zal een onvoorstelbaar aantal andere fouten veroorzaken. De andere mogelijkheid is, dat de apparaatbesturing controleert of de verzoekwachtrij wel ten minste één IOVB bevat en het aan de DOE IO procedure zal terugmelden als hij erachter komt dat dat niet zo is. In dat geval zal de fout met succes gemaskeerd zijn voor de gebruiker, omdat het IOVB opnieuw aangemaakt kan worden. De les, die hieruit geleerd kan worden, is dat er in ieder niveau zoveel mogelijkheden voor foutherkenning en foutherstelling opgenomen moet worden als mogelijk is, waardoor het gevaar van ernstige schade vermeden wordt. Meer voorbeelden voor de foutbehandeling op meerdere niveaus, die langs deze lijnen zijn uitgezet, worden getoond in een beschrijving van de betrouwbaarheidsmechanismen van het HYDRA systeem (Wulf, 1975).

Sommige fouten moeten natuurlijk niet gemaskeerd worden omdat zij een grove fout in een gebruikersprogramma aangeven. Dit zijn geen systeem- maar gebruikersfouten: de verantwoordelijkheid van het besturingssysteem houdt op bij het melden ervan aan de gebruiker. Een voorbeeld is de rekenkundige overloop die door de processor wordt waargenomen en via de BNIB aan de systeemkern wordt gemeld. De interruptroutine kan gemakkelijk het huidige proces voor de betrokken processor als schuldige aanwijzen, waarna het proces afgebroken kan worden met een geschikte foutmelding. (Men kan redeneren dat het maskeren van rekenkundige overloop mogelijk is door bijvoorbeeld in de betrokken plaats het grootst mogelijke aantal te zetten dat weergegeven kan worden. Wij beschouwen deze behandeling echter als ongewenst, omdat het waarschijnlijk andere fouten tot gevolg zal hebben, die de oorspronkelijke fout verduisteren.) Andere fouten die niet gemaskeerd mogen worden zijn schendingen van het geheugen en van de bescherming, die het gevolg zijn van het uitvoeren van gebruikersprogramma's. Die laatste schendingen zouden echter vastgelegd kunnen worden omdat deze opzettelijke pogingen tot het doorbreken van de beveiliging van het systeem kunnen aangeven.

10.7 SAMENVATTING

In dit hoofdstuk hebben wij technieken beschreven voor het verbeteren van de betrouwbaarheid van een besturingssysteem. Daar computers binnendringen in veel gebieden van de menselijke activiteit en omdat er steeds meer op hun werking vertrouwd wordt, zal

hun betrouwbaarheid van het allergrootste belang zijn. De technieken die hier beschreven zijn vormen de basis voor het bereiken van de vereiste betrouwbaarheid, hoewel zij nog steeds aanzienlijk verfijnd worden. Voor wat ons papieren besturingssysteem aangaat, merken we op dat zijn geslaagde structuur goed geschikt is voor het toepassen van foutbehandeling op meerdere niveaus, zoals die in sectie 10.6 werd aangegeven. We verwachten daarom dat door het gebruik van de beschreven technieken een hoge mate van betrouwbaarheid bereikt zal worden.

Ten slotte verwijzen we de lezer, die een nadere studie wil maken van betrouwbaarheid, naar de uitstekende overzichten van Denning (1976), Randall e.a. (1978) en Kopetz (1979).

11 Taakbesturing

In de voorgaande hoofdstukken hebben we besproken hoe een grootschalig besturingssysteem voor algemeen gebruik gemaakt kan worden. Op het moment staat ons besturingssysteem er nog wat verloren bij, omdat we nog niet aangegeven hebben hoe we het kunnen vertellen welke opdrachten we willen laten uitvoeren. De volgende stap is het maken van een koppeling tussen de gebruiker en het besturingssysteem, zodat de gebruiker het systeem kan vertellen wat hij ervan verlangt. De koppeling kan ook gebruikt worden voor het geven van informatie - zoals eventuele hulpbronnen die nodig zijn - over iedere taak, zodat het systeem zijn prestaties op de eerder beschreven manieren kan optimaliseren. Natuurlijk loopt niet alle communicatie tussen de gebruiker en het systeem in dezelfde richting; we moeten ook een aanvullende koppeling maken waarmee het systeem de gebruiker kan vertellen wat het gedaan heeft.

11.1 WAT OPMERKINGEN VAN ALGEMENE AARD

De koppeling tussen de gebruiker en het systeem houdt in dat er een taal moet komen waarmee de communicatie plaats kan vinden. De aard van de taal is zeer afhankelijk van het betrokken besturingssysteem; talen voor systemen met meervoudige toegang verschillen vaak hemelsbreed van talen die voor batchsystemen gebruikt worden. De reden hiervoor is dat de interactieve aard van het werken met de meervoudige-toegangsmethode de gebruiker de gelegenheid geeft zelf de richting van zijn werk te bepalen, en te reageren op systeemhandelingen op het moment dat zij plaatsvinden. In het batchsysteem heeft de gebruiker echter geen controle meer over zijn taak, nadat hij die aan het systeem heeft gegeven. Daarom moet hij vooraf expliciete instructies geven over de te volgen koers voor de handelingen, waarbij hij mogelijk alternatieven aangeeft indien hij niet precies kan overzien welke eventualiteiten er kunnen voorkomen.

Een gevolg van de twee werkwijzen is dat talen, die voor batchverwerking gebruikt worden, meestal complexer en krachtiger zijn dan die, welke bij de meervoudige toegang gebruikt worden. In het tweede geval kan de taal zelf slechts een verzameling commando's

zijn die als een richtlijn voor het besturingssysteem gebruikt kunnen worden. Een dergelijke taal wordt vaak *commandotaal* genoemd, als tegenhanger van de algemenere *taakbesturingstaal* die in een batchsituatie gebruikt wordt. Veel besturingssystemen, zoals VME van ICL en MVS van IBM, proberen binnen één enkele taal de mogelijkheden te geven voor zowel batchbesturing als meervoudige-toegangsbesturing. In deze gevallen merken de gebruikers die de meervoudige toegang gebruiken meestal dat zij slechts een deel van de totale taakbesturingstaal gebruiken.

Het is wenselijk dat zowel commando's als taakbesturingstalen gemakkelijk te gebruiken zijn. We zullen daar later nog op terugkomen.

11.2 COMMANDOTALEN

Het voorbeeld uit figuur 11.1, dat min of meer gebaseerd is op de DEC System-10 monitor, geeft een aantal punten aan die in gedachten gehouden moeten worden bij het ontwerpen van een commandotaal. Het voorbeeld is een afschrift van een korte sessie aan het invoerstation; reacties die door het systeem zijn getypt zijn onderlijnd om ze te onderscheiden van de commando's die door de gebruiker zijn gegeven. De getallen die tussen haakjes staan verwijzen naar de verklarende opmerkingen. (N.B.:

Punten die uit het voorbeeld naar voren komen zijn:

(1) Voor het vermijden van onnodig typwerk zouden zinnige afkortingen toegestaan moeten zijn (zo zijn bijvoorbeeld LOG en EX afkortingen van respectievelijk LOGON (inloggen) en EXECUTE (uitvoeren). Het volledige commando kan altijd ingetypt worden als de gebruiker niet zeker is van een bepaalde afkorting.
(2) De commando's moeten bij voorkeur mnemonisch zijn, zodat zij gemakkelijk onthouden kunnen worden en, indien zij vergeten zijn, gemakkelijk geraden kunnen worden (bijvoorbeeld LOGON, EXECUTE en EDIT).
(3) Bij commando's, die parameters hebben, moeten daarvoor vooraf ingestelde waarden gegeven worden als dat zinnig is. (In het EXECUTE commando is de vooraf ingestelde bestandsnaam bijvoorbeeld de naam van het laatst gebruikte bestand; het vooraf ingestelde I/O apparaat is gedurende de uitvoering het invoerstation.) De nauwkeurige keuze van vooraf ingestelde waarden kan een hoop tijdrovend typwerk besparen.
(4) Het zou mogelijk moeten zijn veel voorkomende handeling met weinig moeite uit te voeren. (EXECUTE is een enkel commando dat de vertaling, het laden en de uitvoering van het genoemde bestand verzorgt.)

```
. LOG                                          (1)
USER# 2167,2167
PASSWORD:                                      (2)
LOGGED ON AT 15.39 5 MAY 74
. EX PROG.ALG                                  (3)
ALGOL:PROG
      ERROR AT LINE 3
      UNDECLARED IDENTIFIER
EXIT
. EDIT PROG.ALG                                (4)
       editing commands terminating with EXIT
. EX                                           (5)
ALGOL:PROG
      NO ERRORS
EXIT
LOADER 5K CORE
EXECUTION
      user types any data for program;
      results of execution typed out
END OF EXECUTION     1.26 SECS
. KJOB                                         (6)
LOGGED OFF USER 2167,2167 at 15:54
RUNTIME = 2.18     CONNECT TIME = 15:10
.
```

Figuur 11.1 *Voorbeeld van een sessie aan een invoerstation. Opmerkingen: (1) verzoek om in het systeem te komen; (2) het afdrukken van het wachtwoord wordt onderdrukt; (3) vertaal het programma in het eerder aangemaakte bestand PROG.ALG en voer het daarna uit; (4) wijzig het programma; (5) vertaal hetzelfde programma en voer het uit; (6) verbreek de verbinding met het systeem*

(5) Een verstandig gekozen stel afspraken kan het intypen verminderen. (Een bestandsnaam waarvan het tweede deel ALG is, wordt herkend als een Algol programma.)
(6) De vorm waarin de commando's gegeven worden zou zo vrij mogelijk moeten zijn (bijvoorbeeld het aantal spaties tussen twee elementen van een commando is niet van belang).
(7) Het moet altijd duidelijk zijn of het systeem al dan niet in staat is het volgende commando te ontvangen (dit wordt aangegeven door de punt aan het begin van iedere lijn).
(8) Het systeem zou de gebruiker voldoende informatie moeten geven omtrent hetgeen er gebeurt, maar niet zoveel dat hij door de bomen het bos niet meer ziet.

We zullen niet verder uitweiden over commandotalen, uitgezonderd nog de opmerking dat veel van de bovenstaande punten ook voor taakbesturingstalen gelden.

11.3 TAAKBESTURINGSTALEN

In de eerste batchverwerkende systemen, zoals het IBSYS systeem voor de IBM 7090 serie, werd de taakbesturing verkregen door het voegen van *besturingskaarten* tussen de kaarten met het programma en de gegevens. De besturingskaarten hadden een speciaal herkenningsteken (bij IBSYS een $ in kolom 1) en werden door het besturingssysteem geïnterpreteerd als instructies over wat er met de volgende kaarten moest gebeuren. Er waren drie nadelen verbonden aan deze werkwijze:

(1) De volgorde van de handelingen kan niet veranderd worden naar aanleiding van het al dan niet slagen van de voorgaande stappen.
(2) De besturingsinformatie wordt over de gehele invoer versnipperd, wat niet logisch lijkt en wat de kans op fouten verhoogt.
(3) Er is een speciale tekencombinatie op de besturingskaarten nodig voor de herkenning daarvan; die combinatie mag daarom nergens in het programma of de gegevens voorkomen. (Een manier om dit te omzeilen is het voorzien in een speciale besturingsopdracht, die het herkenningsteken van alle erop volgende besturingskaarten verandert in een andere tekencombinatie. Deze voorziening is in het OS/360 JCL aanwezig.)

In enkele systemen (bijvoorbeeld het DEC System-10 batch) bestaat dit soort taakbesturing nog steeds. Veel systemen vereisen echter in navolging van de Atlas dat alle informatie over de taakbesturing aan het begin van de taak in een *taakomschrijving* gezet moet worden. De taakomschrijving is in feite een programma dat in een

taakbesturingstaal geschreven is en dat de besturing van de hele taak door het computersysteem bestuurt.

De besturingsinformatie die in de taakomschrijving gegeven wordt kan als volgt ingedeeld worden.

(1) Verantwoordingsinformatie

Deze bestaat meestal uit de naam van de taak en de identiteit van de gebruiker. Die identiteit kan vergeleken worden met de lijst van gebruikers die te goeder trouw zijn. Eventueel gebruik van hulpbronnen, waarvoor betaald moet worden, kan bij de goede rekening gedebiteerd worden.

(2) Informatie voor de werkindeling

Deze bestaat gewoonlijk uit een specificatie van het maximale gebruik van de diverse hulpbronnen door de taak. Bij systemen zoals het OS/360 en zijn afstammelingen, waarbij de taken in taakstappen onderverdeeld worden, kan een specificatie van de hulpbronnen voor iedere stap nodig zijn. De informatie die nodig is, is die welke door de werkindeler gebruikt kan worden voor:

(a) de beslissing wat de volgende taak is die gestart moet worden;
(b) het toewijzen van prioriteiten;
(c) het vermijden van het vastlopen van het systeem;
(d) het beëindigen van taken die meer gebruik van hulpbronnen gaan maken dan toegestaan is.

Typerende specificaties voor hulpbronnen zijn onder andere: de hoeveelheid gebruikt geheugen, de processortijd en de grenzen die aan de uitvoer gesteld worden.

Verdere informatie die voor het maken van een werkindeling verstrekt zou kunnen worden, is een verzoek om een taak een bepaalde prioriteit te geven, of een verzoek dat een taak na een bepaalde andere taak wordt uitgevoerd. Dergelijke informatie over de volgorde is nuttig in gevallen waarbij een taak afhankelijk is van een bestand dat achtergelaten wordt door een andere taak.

(3) I/O informatie

Dit is de informatie die in systemen gebruikt wordt die op stromen zijn gebaseerd. Hierbij worden stromen aan de fysieke I/O apparatuur gekoppeld (de lezer die zijn kennis over stromen wat wil ophalen verwijzen wij naar sectie 6.1). In niet op stromen gebaseerde systemen wordt dit deel van de taakomschrijving gebruikt voor de verzoeken om toewijzing van I/O apparatuur aan de taak. Er kunnen onder deze kop ook specificaties staan over tekencodes en de grootte van blokken.

(4) Procedurele informatie

Dit is het deel van de taakomschrijving dat feitelijk aan het besturingssysteem vertelt wat de gebruiker ervan verwacht. Sommige taakbesturingstalen kunnen slechts het geven van een eenvoudige opeenvolging van richtlijnen toestaan; andere kunnen voorzien in krachtiger faciliteiten zoals hieronder beschreven.

Figuur 11.2 geeft als voorbeeld een Atlas-achtige taakomschrijving weer. Hierin kunnen die vier soorten informatie gemakkelijk onderscheiden worden. Andere taakbesturingstalen, zoals die in sommige IBM systemen, vermengen verschillende soorten informatie in een enkele opdracht.

```
                                Opmerkingen

JOB AML 123 SAMPLE              kop en verantwoordings-
                                informatie

CORE 15 PAGES 25        |
                        |
TIME 200                |       informatie voor de werkindeler
                        |
AFTER AML 123 SORT      |

INPUT 1 = CDR           |       I/O informatie. De aanhalings-
                        |       tekens worden gebruikt voor het
INPUT 2 = "MYDATA"      |       onderscheiden van bestands- en
                        |       apparaatnamen
OUTPUT 1 = LPT          |

COMPILE FORTRAN         |       procedurele informatie. De
                        |       bestandsnaam voor het vertaalde
EXECUTE                 |       programma is vooraf ingesteld.

END

      programmakaarten

****                            beëindiger van de taak
```

Figuur 11.2 *Atlas-achtige taakomschrijving*

Hoe krachtig een taakbesturingstaal is wordt voor het grootste deel bepaald door zijn faciliteiten voor het aangeven van procedurele informatie. Een aantal van deze faciliteiten wordt hieronder besproken.

(1) *Macro's*. De gebruiker kan door het geven van macro's samengestelde handelingen met een commando specificeren. Iedere opeenvolging van taakbesturingsopdrachten kan gedefinieerd worden door een benoemde *macro* (of *katalogusprocedure*). Deze kan simpelweg aangeroepen worden door het aanhalen van de naam van de macro met de erbij behorende parameters. Zo kan bijvoorbeeld de opeenvolging van de opdrachten, die nodig zijn voor het vertalen, laden en uitvoeren van een Fortran programma, gedefinieerd worden als een macro met de naam 'FORTRAN'; deze macro kan gebruikt worden met parameters die de bestandsnaam van het bronprogramma en de bestemming van de uitvoer aangeven. Het uitbreiden van een macro of de vervanging van een parameter wordt gedaan door het programma (interpreter) dat de taakomschrijving stap voor stap omzet voordat het uitgevoerd wordt; het mechanisme is vergelijkbaar met hetgeen gebruikt wordt in een assembler. De meeste besturingssystemen geven de beschikking over een 'macrobibliotheek' voor de meest voorkomende activiteiten. Veel besturingssystemen geven de gebruikers ook de mogelijkheid hun eigen bibliotheek op te bouwen om aan hun specifieke behoefte te voldoen.

(2) *Besturingsstructuur*. Taakbesturingstalen, die slechts eenvoudige reeksen van opeenvolgende opdrachten toestaan, zijn niet geschikt voor taken waarvan de acties die ondernomen moeten worden afhangen van het resultaat van voorgaande stappen. Diverse talen bevatten de een of andere vorm van de if opdracht, zodat delen van een taakomschrijving alleen uitgevoerd kunnen worden indien er aan bepaalde voorwaarden voldaan wordt. De voorwaarden waarop de controles kunnen plaatsvinden zijn meestal het resultaat van een eerdere stap; typerende voorbeelden zijn het optreden van een fout, de inhoud van een bepaalde boodschap, of het voorkomen van een bepaalde waarde die achtergelaten is in een machineregister. Enkele talen hebben ook een lusmogelijkheid (Engels: loop), waardoor delen van een taak meer dan eens uitgevoerd kunnen worden. Het voorzien in voorwaardelijke constructies en herhalingsconstructies zorgt ervoor dat besturingstalen steeds meer op hogere programmeertalen gaan lijken. Er is hoegenaamd geen reden om aan te nemen dat ze zich niet verder in die richting gaan ontwikkelen doordat mogelijkheden als variabelen, blokstructuren en procedures opgenomen worden. Deze met hogere talen vergelijkbare benadering is al toegepast op een paar systemen (zoals de ICL 2900 serie).

(3) *Parallelliteit*. Enkele taakbesturingstalen (bijvoorbeeld die in UNIX) hebben de mogelijkheid om aan te geven dat bepaalde programmadelen parallel uitgevoerd kunnen worden. Dergelijke

informatie kan door de werkindeler gebruikt worden voor het starten van een taak als een stel parallel in plaats van opeenvolgend lopende processen. Naarmate er meer systemen met meerdere processoren beschikbaar komen, waarbij de parallelliteit reële voordelen kan opleveren, zal deze faciliteit mogelijk wat algemener ingang vinden.

Een mogelijke toekomstige ontwikkeling trekt de overeenkomsten met hogere programmeertalen nog verder door via het invoeren van taakbesturingstalen die in hun geheel of in stukjes vertaald kunnen worden naar ieder systeem. Dit zou aanzienlijk bijdragen tot de machine-onafhankelijkheid voor het uitvoeren van taken; op het moment wordt de machine-onafhankelijkheid, die verkregen wordt op het programmeerniveau, teniet gedaan door de verschillende manieren waarop de taken aan de verschillende machines gepresenteerd moeten worden. Belemmeringen die een dergelijke ontwikkeling in de weg staan, komen voort uit de problemen die er zijn om de verschillende systemen te voorzien van dezelfde koppelingen naar de buitenwereld (Dakin, 1975).

Voor zover het ons papieren besturingssysteem aangaat, zullen we niet proberen een bepaalde taakbesturingstaal te ontwerpen. In plaats daarvan merken we alleen maar op dat, onafhankelijk van welke taal we gaan gebruiken, deze de hiervoor beschreven eigenschappen zou moeten bezitten en zo ontworpen moet zijn dat de in sectie 11.2 opgesomde eigenschappen, die vanuit het standpunt van de gebruiker wenselijk zijn, aanwezig zijn.

11.4 DE TAKENPOT

We hebben in sectie 8.4 opgemerkt dat taken die op het begin van hun uitvoering wachten, opgeslagen zijn in een *takenpot*. Iedere ingang in de takenpot staat voor één enkele taak en bevat informatie daarover die afgeleid is uit de taakomschrijving. Deze informatie is in feite een gecodeerde vorm van de informatie die in de taakomschrijver staat. De enige wijziging is, dat in een systeem waar spooling wordt toegepast de namen van alle invoerapparatuur vervangen worden door de namen van de bestanden waarnaar de invoer overgebracht moet worden. Dit overbrengen kan door het vertaalprogramma voor de taakomschrijver zelf verzorgd worden, maar het is ook mogelijk dat het vertaalprogramma deze taak delegeert naar aparte 'invoerspoolers' (zie sectie 6.6). Deze worden geactiveerd door het geven van een verhoogsignaal aan de bijbehorende seinpalen. In beide gevallen meldt het vertaalprogramma aan de werkindeler wanneer het uitvoeren van een taak in de takenpot voltooid is.

De takenpot kan de vorm hebben van één enkele lijst, of van een aantal lijsten, waarin iedere lijst de taken met bepaalde

eigenschappen bevat. Deze tweede structuur kan de werkindeler tot hulp zijn bij de keuze welke taak er vervolgens uitgevoerd moet worden.

De omvang van de takenpot kan zo groot zijn dat deze onmogelijk in het geheugen opgeslagen kan worden. De opslag op schijf kan echter veel extra werk voor de werkindeler met zich meebrengen wanneer die moet gaan zoeken naar die taak, die het beste uitgevoerd kan gaan worden. Een mogelijk compromis is het bewaren van afgekorte versies van de ingangen van de takenpot in het geheugen, terwijl de volledige versies op de schijf staan. De afgekorte versies zullen alleen informatie bevatten die voor de werkindeler van belang is bij het kiezen van de volgende taak; informatie over bijvoorbeeld invoerbestanden kan weggelaten worden.

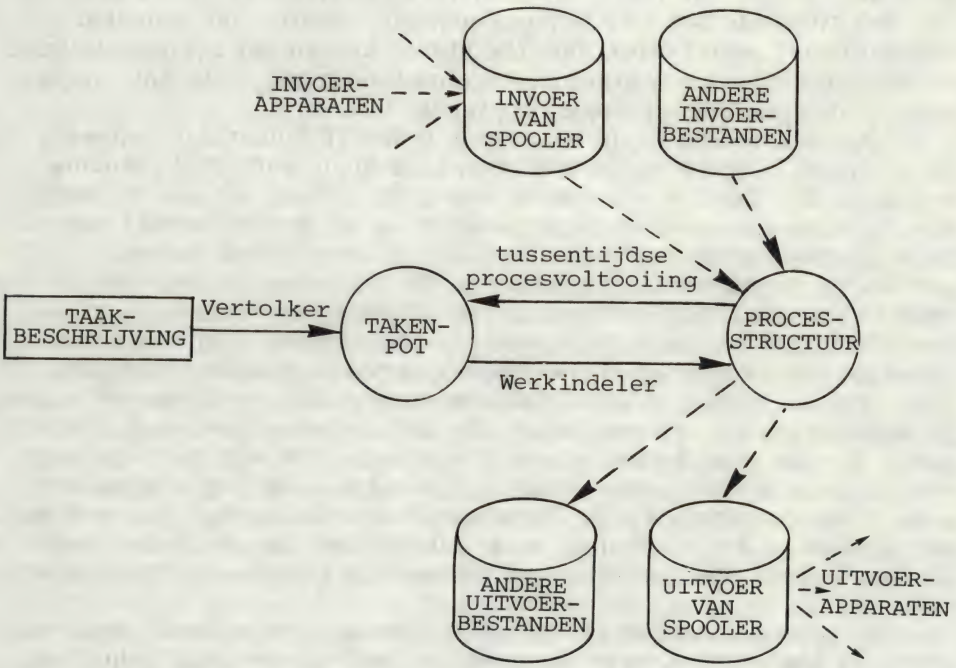
11.5 MELDINGEN VAN HET SYSTEEM

De verhouding tussen de gebruiker en het besturingssysteem hangt niet alleen af van het gemak waarmee de gebruiker aan het systeem kan vertellen wat hij ervan verlangt, maar ook van de kwaliteit van de informatie die hij van het systeem ontvangt over wat het gedaan heeft. De informatie die door de huidige machines verstrekt wordt, in het bijzonder als er een fout in de uitvoering van de taak is opgetreden, varieert aanzienlijk in kwaliteit, detaillering, relevantie en begrijpelijkheid. In het ideale geval hangt de mate van detaillering van het lot van de taak af; als de taak goed voltooid wordt, dan is een korte verantwoording van de hulpbronnen, die in rekening gebracht moeten worden, waarschijnlijk voldoende; indien de taak echter niet goed verloopt, is er een volledige beschrijving van de oorzaak nodig. In ieder geval moeten de foutmeldingen in begrijpelijke taal (meestal Engels) gesteld zijn, zodat ze voor de normale gebruiker te begrijpen zijn. Te vaak komt het voor dat de gebruiker opgezadeld wordt met een lading schijnbaar gebrabbel, of in het geval van een storing met een hexadecimale afdruk!

In een situatie waarin meervoudige toegang mogelijk is, moet de gebruiker informatie kunnen opvragen over de huidige toestand van het systeem wat betreft het aantal gebruikers dat verbinding heeft met het systeem, welke hulpbronnen er beschikbaar zijn, enzovoort. Deze informatie maakt het hem mogelijk in te schatten wat voor respons hij kan verwachten en maakt het hem ook mogelijk te beslissen of het al dan niet zin heeft met een sessie te beginnen of door te gaan.

11.6 DE GANG VAN EEN TAAK DOOR HET SYSTEEM

Figuur 11.3 geeft beknopt de stadia weer die een taak in ons papieren besturingssysteem doorloopt, vanaf de invoering tot de beëindiging. De doorgetrokken lijnen geven de diverse metamorfoses van de taak weer vanaf de taakomschrijving, via het opgenomen worden in de takenpot, naar de opeenvolging van de processen. De gestippelde lijn geeft de gang van de gegevens en van de programma's weer die bij de taak betrokken zijn. Als er geen spooling toegepast wordt, kunnen de wachtrijen voor de in- en uitvoer weggelaten worden uit het diagram, en de gegevensoverdrachten kunnen dan weergegeven worden als direct van of naar de randapparaten. De cirkelbeweging tussen de takenpot en de processtructuur is bedoeld om te benadrukken dat er meerdere processen betrokken kunnen zijn bij het uitvoeren van één enkele taak. Binnen de processtructuur bewegen de processen zich tussen de drie toestanden: geblokkeerd, uitvoerbaar en lopend, zoals eerder in figuur 8.4 is weergegeven.



Figuur 11.3 *De voortgang van een taak*

Tot slot

In het vorige hoofdstuk hebben we de bouw van ons papieren besturingssysteem voltooid door het te voorzien van een koppeling met de buitenwereld. Door middel van het beschrijven van de voortgang van een taak tussen start en finish, hebben we ook een overzicht van het systeem gegeven.

Het doel van de bouw en de beschrijving van het papieren besturingssysteem was tweeledig. De eerste doelstelling is geweest het als een voertuig te gebruiken voor het illustreren van de mechanismen waardoor een besturingssysteem de functies kan vervullen die de gebruiker ervan verwacht. De tweede doelstelling is het steunen van het steeds bekendere feit geweest, dat besturingssystemen betrouwbaar, gemakkelijk onderhoudbaar en relatief foutloos kunnen worden, indien zij een logische en samenhangende structuur vertonen. We komen hier zo meteen op terug.

Voor wat het eerste doel aangaat is het te hopen dat de lezer, die dit punt bereikt heeft, een goed idee heeft van wat besturingssystemen zijn, wat ze kunnen en hoe ze dat doen. Hij zou de aard van de problemen moeten begrijpen die hij bij het ontwerpen van besturingssystemen ontmoet en hij zou de technieken moeten begrijpen om die problemen te overwinnen. In het korte bestek van dit boek is het natuurlijk onmogelijk een uitgebreid overzicht te geven van alle problemen en alle oplossingen; er kunnen alleen de meest voorkomende problemen en de meest gebruikte oplossingen beschreven worden. Er zijn onvermijdelijk systemen die, vanwege een andere gebruiksomgeving of beperkingen in de hardware, problemen veroorzaken die hier niet behandeld zijn; ook zullen er vele systemen zijn waarin de problemen, die we hier genoemd hebben, op een andere manier zijn opgelost. We hopen echter dat het materiaal dat we hebben laten zien een voldoende basis zal vormen voor de lezer, zodat hij van daaruit zijn kennis, naar gelang zijn interesse, kan uitbreiden.

Het tweede punt dat hierboven genoemd is, aangaande de structuur van besturingssystemen, verdient een nadere bespreking. We hebben op papier een systeem gemaakt dat keurig gelaagd is, waarbij iedere laag alleen van de eronder liggende lagen afhankelijk is. We moeten echter toegeven, dat de goden ons goed gezind waren toen we ons systeem bouwden. Ten eerste werden we niet afgeleid door de problemen die het werken in een grote groep met zich meebrengt. Ten tweede werden we niet door een verzoek van de klant

verplicht na het eerste ontwerpstadium nieuwe mogelijkheden aan het systeem toe te voegen. Ten derde, en dat is misschien wel het belangrijkste, hoefden we het systeem niet op een echte machine te laten werken. Men moet zich daarom de volgende vraag stellen: in hoeverre is het mogelijk de structuur van het papieren systeem naar de werkelijkheid over te brengen?

De eerste twee hindernissen - die van de grote projectgroepen en het veranderen van de specificaties - kunnen omzeild worden door een goed management. De derde hindernis - de onverzoenlijkheid van de machine-architectuur - is moeilijker te nemen, zeker op korte termijn. De machine-architectuur botst op vele plaatsen met het systeem, maar de gebieden die direct beïnvloed worden zijn interruptbesturing, I/O en geheugenbescherming. Het is de moeite waard die één voor één te bekijken.

Zoals in hoofdstuk 4 werd aangegeven kan de complexiteit van de interruptbesturing aanzienlijk variëren, al naar gelang de hulp die door de hardware wordt geboden. Deze hulp is in het bijzonder van belang bij de identificatie van interrupts en bij het vaststellen van de prioriteiten. Het DEC System-10, dat in een hardware interruptmechanisme voorziet, waarbij apparaten iedere prioriteit van de in totaal zeven kunnen krijgen, geeft bijvoorbeeld meer hulp dan de IBM 370, waarbij de prioriteit voor het grootste deel van tevoren vastgesteld is en waarvan die prioriteiten alleen met selectieve softwaremaskering gedeeltelijk te wijzigen zijn. Beide machines geven echter voldoende informatie voor het gemakkelijk vaststellen van de bron en de aard van een interrupt.

Het frame voor de I/O afhandeling, dat in hoofdstuk 6 opgesteld werd, kan behoorlijk door de war geschopt worden vanwege de druk die er door bepaalde I/O architectuur op gelegd wordt. Het kader is uitstekend geschikt voor bouwwijzen die op een eenvoudig bussysteem zijn gebaseerd, maar moet mogelijk gewijzigd worden in gevallen (zoals de IBM 370) waarbij een hiërarchische structuur van kanalen, besturingseenheden en apparaten gebruikt wordt. In de grote CDC machines worden de I/O functies uitbesteed aan speciale processoren voor de randapparatuur, en de apparaatbesturing wordt door de processoren voor de randapparatuur uitgevoerd, in plaats van door een centrale verwerkingseenheid. Invoerstations worden vanwege hun speciale besturingsfuncties in de meeste systemen als een apart geval behandeld. Zo worden in het DEC System-10 de buffers voor de invoerstations bijvoorbeeld uit de geheugenruimte voor het besturingssysteem toegewezen, en niet uit de ruimte die bij de individuele gebruikersprocessen hoort (wat wel zo is voor de andere apparaten). De reden hiervoor ligt in het feit dat een proces niet uit het hoofdgeheugen gegooid mag worden wanneer het met I/O handelingen bezig is; zijn buffers zouden namelijk overschreven kunnen worden. Omdat buffers voor invoerstations altijd in gebruik kunnen zijn, volgt daaruit dat zij, indien zij uit de gebruikersruimte werden toegewezen, nooit uit het systeem gegooid zouden kunnen worden.

De architectonische eigenschap, die waarschijnlijk de grootste invloed heeft op de structuur van een besturingssysteem, is het mechanisme voor de geheugenbescherming. Het volledig in de praktijk brengen van de gelaagde structuur van ons papieren besturingssysteem betekent dat er een beschermingssysteem moet bestaan dat daarvan een weerspiegeling moet kunnen geven. Omdat het segment de logische eenheid voor de adresruimte is, lijkt het de aangewezen beschermingseenheid; het lijkt ook de eenheid voor het maken van een willekeurig aantal beschermingsdomeinen. Een machine die ver van dit ideaal afwijkt is de IBM 370. Bij die machine wordt de bescherming toegepast op fysieke geheugenblokken van 2K-byte; de verwarring tussen logische en fysieke verdelingen in het geheugen wordt nog groter gemaakt doordat het paginaveld van een adres over mag lopen naar het segmentveld. Diverse machines, waaronder de Burroughs 6700 en de Honeywell 645, passen bescherming op segmenten toe, maar er zijn er maar weinig die meer dan twee of drie beschermingsdomeinen hebben. De bekendste opstelling is het tweedomeinen 'supervisor-gebruiker' systeem dat weergegeven wordt door de IBM 370 en vele microprocessoren. Meervoudige domeinen zijn er in de CYBER 180 en de ICL 2900 serie, maar dat is een vast aantal en zij brengen veel extra werk voor de besturing met zich mee. De problemen voor het implementeren van meervoudige domeinen tegen lage kosten zijn bij lange na nog niet opgelost; de toepassing van 'functies', zoals beschreven in hoofdstuk 9, is misschien nog de meest veelbelovende oplossing. Ondanks de onvolkomenheden in de huidige architectuur kunnen we verwachten dat nieuwe beschermingsmechanismen het binnenkort mogelijk zullen maken dat de gelaagde structuur wordt toegepast. Onderwijl kunnen we structuur beschouwen als een legitieme doelstelling, waarvoor de middelen ontbreken om deze te bewerkstelligen.

Wat voor ontwikkelingen kunnen we verder nog verwachten? Koffiedik kijken is een riskante bezigheid in een zo snel veranderend gebied als dit. We zullen het echter kort aanroeren met het risico door de gebeurtenissen terechtgewezen te worden. Op de korte termijn zullen de ontwikkelingen zich waarschijnlijk concentreren op het verbeteren van de betrouwbaarheid van besturingssystemen, op het vereenvoudigen van het overbrengen van het systeem van de ene naar de andere machine en op het verhogen van het gebruiksgemak door het voorzien in krachtiger en flexibeler taakbesturingstalen. Op de langte termijn zijn de twee factoren, waarvan we verwachten dat zij de meeste invloed zullen hebben op de aard en de rol van besturingssystemen, de komst van microprocessoren en het voorzien in betere voorzieningen voor de telecommunicatie.

Microprocessoren bieden een nieuwe vorm van verwerking - kleine afzonderlijke eenheden die erg goedkoop zijn, in tegenstelling tot de grote monolitische dure eenheden van de afgelopen 30 jaar. Een trend is al duidelijk zichtbaar: het verdelen van verwerkingsfaciliteiten naar de plaatsen waar zij het meest nodig zijn, in plaats van het samenbrengen daarvan op één plaats. Daar de kosten van processoren, geheugen en secundaire opslagmogelijkheden snel

verminderen, is het economisch aantrekkelijk geworden een computersysteem aan één enkele taak te zetten, of dat nu het maken van salarisstroken, de procesbesturing, het onderwijzen, het bijhouden van het huishoudboekje, of wat dan ook is. Het toewijzen van hulpbronnen, een van de belangrijkste functies van het besturingssysteem van een mainframecomputer, wordt in zo'n systeem triviaal, omdat alle hulpbronnen die nodig zijn altijd beschikbaar zijn.

Systemen met maar één taak kunnen volledig onafhankelijk zijn, of ze kunnen aan elkaar gekoppeld zijn met behulp van overdrachtlijnen voor de gegevens. In dat tweede geval kan de informatie, die op de ene plaats opgeslagen ligt, overgeseind worden om op een andere plaats gebruikt te worden, of kan werk, dat teveel is voor de capaciteit van één enkele installatie, voor verwerking overgedragen worden aan een andere. De mogelijkheid om semi-onafhankelijke installaties tot één groot netwerk aan elkaar te koppelen, opent het perspectief dat aan ver uiteenliggende gebruikers(groepen) een enorme verwerkingskracht en een enorm gegevenbestand ter beschikking gesteld kan worden. De functie van een besturingssysteem in een dergelijke situatie omvat onder andere: de overdracht en controle op juistheid van mededelingen, de verdeling van de last, en het handhaven van de samenhang van de informatie die in diverse vormen op verschillende plaatsen ligt opgeslagen.

Een andere ontwikkeling, die mogelijk is in gebieden waar grote hoeveelheden verwerkingskracht vereist blijven, is de vervanging van de mainframecomputer door een verzameling onderling verbonden microprocessoren. Een voordeel hiervan is dat ieder proces zijn eigen processor ter beschikking heeft, waardoor de parallelliteit in het systeem verhoogd wordt en de noodzaak tot schakelen tussen processen verlaagd wordt. Nog een voordeel is het mogelijk toenemen van de betrouwbaarheid als gevolg van de onderlinge uitwisselbaarheid van processoren. Er zijn echter diverse moeilijkheden die overwonnen moeten worden voordat een dergelijk systeem gerealiseerd kan worden. Eén van de belangrijkste daarvan heeft betrekking op de organisatie van het geheugen: moet iedere processor zijn eigen geheugen hebben, moeten ze allemaal een gemeenschappelijk geheugen gebruiken, of moet het een combinatie van de twee zijn? Een gemeenschappelijk geheugen brengt moeilijkheden voor de toegang met zich mee, terwijl het gebruik van aparte geheugens de eis met zich meebrengt dat er een gemakkelijke gegevensoverdracht mogelijk is. Een gemengd geheugen vereist de oplossing van beide soorten problemen, samen met een oplossing voor het verdelen en implementeren van een geschikte adresvertaling. Andere problemen van architectonische aard zijn het sturen van een externe interrupt naar de juiste processor en het voorzien in een mechanisme voor de communicatie tussen de processoren. Op het niveau van het besturingssysteem zijn er problemen aangaande de herkenning van mogelijke parallelliteit, de toewijzing van processen aan processoren, de communicatie tussen processen die geen geheugen gemeenschappelijk hebben, het delen van hulpbronnen, en de foutherkenning en herstelling. Aan al deze problemen wordt hard gewerkt en het is te

verwachten dat levensvatbare multi-microprocessorsystemen binnen een paar jaar werkelijkheid worden.

Tot slot zou ik de lezer op het hart willen drukken dit boek in geen enkel opzicht te beschouwen als het laatste woord over besturingssystemen. Er zijn vele tijdschriften en boeken die een frisse blik geven op het hier besproken materiaal, of die de lezer nieuwe gebieden tonen. Ik hoop dat de verwijzingen die op de volgende bladzijden staan een voldoende aanknopingspunt zullen bieden.

Appendix: Monitors

De algemene structuur van een monitor (Hoare, 1974) met gebruik van een Pascal-achtige syntaxis, is:

```
var m: monitor
begin
    het declareren van lokale variabelen die het gedeelde
    object weergeven
    het declareren van procedures voor toegang tot de
    lokale variabelen
    het initialiseren van de lokale variabelen
end
```

De lokale variabelen zijn buiten de monitor ontoegankelijk; na hun initialisering kan er alleen door de monitorprocedures zelf mee gewerkt worden. Om gelijktijdige veranderingen van lokale variabelen door meerdere processen te voorkomen, zijn de monitorprocedures als elkaar wederzijds uitsluitende kritische sectoren geïmplementeerd. Wederzijdse uitsluiting is door het vertaalprogramma gegarandeerd, die de juiste synchronisatiehandelingen (zoals *passeer*- en *verhoog*handelingen op seinpalen) in het vertaalde programma aanbrengt. (Voor de lezers die met abstracte gegevenssoorten bekend zijn: een monitor kan als een abstract gegevensobject beschouwd worden dat veilig door meerdere processen gedeeld kan worden.)

Alhoewel de wederzijdse uitsluiting van monitorprocedures door het vertaalprogramma gegarandeerd is, blijven andere vormen van synchronisatie de verantwoordelijkheid van de programmeur. De synchronisatiehandelingen, die door Hoare voorgesteld worden en die in de meeste implementaties van monitoren toegepast worden, zijn geen *passeer*- en *verhoog*handelingen op seinpalen, maar vergelijkbare handelingen op *voorwaarden* (Engels: *conditions*). Net zoals bij een seinpaal heeft een voorwaarde twee handelingen die erop uitgevoerd kunnen worden - één wordt voor het vertragen, de ander voor het hervatten van een proces gebruikt. We zullen er met respectievelijk *pauze* en *vervolg* naar verw[illegible]

De definities van *pauze* en *vervolg* zijn:

pauze (voorwaarde):	schort de uitvoering van het aanroepende proces op
vervolg (voorwaarde):	hervat de uitvoering van een willekeurig proces dat na een *pauze* op dezelfde voorwaarde is opgeschort. Indien er meerdere van dergelijke processen zijn, kies er dan een van; indien er geen dergelijk proces is, doe dan niets

De *pauze*handeling heft de uitsluiting op de monitor op - anders zou geen enkel ander proces in staat zijn de monitor in te gaan voor het uitvoeren van een *vervolg*. Overeenkomstig hiermee heft *vervolg* de uitsluiting op en draagt die over aan het proces dat hervat is. Het hervatte proces moet daarom in staat zijn te lopen voordat enig ander proces de monitor binnen kan gaan.

Alhoewel voorwaarden en seinpalen voor hetzelfde doel gebruikt worden, zijn er de volgende beduidende verschillen:

(a) Een voorwaarde heeft, in tegenstelling tot een seinpaal, geen waarde. Men kan het zich voorstellen als een wachtrij waaraan een proces wordt toegevoegd bij het uitvoeren van *pauze*, en waaruit een proces wordt verwijderd indien een andere proces een *vervolg* uitvoert.
(b) Een *pauze*handeling heeft *altijd* een vertraging in de uitvoer van een proces tot gevolg. Dit in tegenstelling tot de *passeer*handeling, die alleen voor vertraging zorgt indien de waarde van de seinpaal nul is.
(c) Een *vervolg*handeling heeft alleen maar gevolgen indien een proces opgeschort is op dezelfde voorwaarde. Een *verhoog*handeling heeft daarentegen altijd een gevolg, en wel het verlagen van de waarde van de seinpaal. Daarom worden *verhoog*handelingen 'onthouden' (in de waarde van de seinpaal) en *vervolg*handelingen niet.

Als voorbeeld voor het gebruik van voorwaarden ten behoeve van de synchronisatie geven we hier de details van de buffermonitor waarnaar in sectie 3.4 verwezen werd.

```
var buffer: monitor;
begin
    var B: array [0...N-1] of item;      {ruimte voor N onderdelen}
        nextin,nextout: 0...N-1;         {bufferwijzer}
        nonfull,nonempty: condition;     {voor synchronisatie}
        count: 0...N;                    {aantal onderdelen in buffer}
```

```
procedure deposit(x: item);
begin
    if count = N then cwait(nonfull);   {vermijd overlopen}
    B[nextin] := x;
    nextin := (nextin+1) mod N;
    count := count + 1;                 {één onderdeel meer in buffer}
    csignal(nonempty)                   {hervat een wachtende afnemer}
end;

procedure extract(var x: item);         {vermijd onder nul raken}
begin
    if count = 0 then cwait(nonempty);
    x := B[nextout];
    nextout := (nextout + 1) mod N;
    count := count - 1;                 {één onderdeel minder in buffer}
    csignal(nonfull)                    {hervat een wachtende producent}
end

    nextin := 0; nextout := 0;
    count := 0                          {buffer is leeg bij de aanvang}
end
```

(De termen *cwait* en *csignal* staan voor onze termen *pauze* en *vervolg*.)

Literatuur

Engelstalig

Alagic, S., and Arbib, M.A. (1978), *The Design of Well-Structured and Correct Programs*, Springer-Verlag, New York.

Andrews, G.R., and Schneider, F.B. (1983), Concepts and notations for concurrent programming, *Computing Surveys*, 15, 3-43.

Avizienis, A. (1977), Fault-tolerant Computing: Progress, Problems and Prospects, *Proceedings IFIPS 77*, North-Holland, Amsterdam, p. 405.

Brooks, F.P. (1975), *The Mythical Man-Month*, Addison-Wesley, Reading, Mass.

Coffman, E.G., and Denning, P.J. (1973), *Operating System Theory*, Prentice-Hall, Englewood Cliffs, N.J.

Coffman, E.G., and Kleinrock, L. (1968), Computer scheduling methods and their countermeasures, *Proceedings AFIPS Spring Joint Computer Conference*, 32, 11-21.

Coffman, E.G., Elphick, M., and Shoshani, A. (1971), System deadlocks, *Computing Surveys*, 3, 67-78.

Colin, A.J.T. (1971), *Introduction to Operating Systems*, MacDonald-Elsevier, London and New York, p. 36.

Corbató, F.J., Merwin-Dagget, M., and Daley, R.C. (1962), An experimental time sharing system, *Proceedings AFIPS Spring Joint Computer Conference*, 21, 335-344.

Courtois, P.J., Heymans, R., and Parnas, D.L. (1971), Concurrent control with 'readers' and 'writers', *Comm. ACM*, 14, 667-668.

Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. (1972), *Structured Programming*, Academic Press, London and New York.

Dakin, R.J. (1975), A general control language: structure and translation, *Computer Journal*, 18, 324-332.

Daley, R.C., and Dennis, J.B. (1968), Virtual memory, processes, and sharing in MULTICS, *Comm. ACM* 11, 306-312.

Daley, R.C., and Neumann, P.G. (1965), A general-purpose file system for secondary storage, *Proceedings AFIPS Fall Joint Computer Conference*, 27, 213-229.

Denning, P.J. (1968), The working set model for program behaviour, *Comm. ACM*, 11, 323-333.

Denning, P.J. (1970), Virtual memory, *Computing Surveys*, 2, 153-189.

Denning, P.J. (1971), Third generation computer systems, *Computing Surveys*, 3, 175-216.

Denning, P.J. (1976), Fault-tolerant operating systems, *Computing Surveys*, 8, 359-389.

Dennis, J.B., and Van Horn, C. (1966), Programming semantics for multi-programmed computations, *Comm. ACM*, 9, 143-155.

Digital Equipment Corporation (1973), *DEC System-10 Assembly Language Handbook*, 3rd edition.

Dijkstra, E.W. (1965), Cooperating sequential processes. In *Programming Languages* (ed. F. Genuys), Academic Press, New York (1968).

Dijkstra, E.W. (1968), The structure of the T.H.E. multiprogramming system, *Comm. ACM*, 11, 341-346.

Dijkstra, E.W. (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J.

England, D.M. (1974), The capability concept mechanism and structure in System-250, *IRIA International Workshop on Protection in Operating Systems*, Rocquencourt, 63-82.

Evans, D.C., and Leclerc, J.Y. (1967), Address mapping and the control of access in an interactive computer, *Proceedings AFIPS Spring Joint Computer Conference*, 30, 23-32.

Fabry, R.S. (1974), Capability based addressing, *Comm. ACM*, 17, 403-412.

Floyd, R.W. (1967), Assigning meanings to programs, *Proc. Symposium in Applied Maths*, 19, American Math. Soc.

Foster, C.C. (1970), *Computer Architecture*, Van Nostrand Reinhold, New York, pp. 146-152.

Frank, G.R. (1976), Job control in the MU5 operating system, *Computer Journal*, 19, 139-143.

Fraser, A.G. (1969), Integrity of a mass storage filing system, *Computer Journal*, 12, 1-5.

Graham, G.S., and Denning, P.J. (1972), Protection, principles and practice, *Proceedings AFIPS Spring Joint Computer Conference*, 40, 417-429.

Graham, R.M. (1968), Protection in an information processing utility, *Comm. ACM*, 11, 365-369.

Habermann, A.N. (1969), Prevention of system deadlock, *Comm. ACM*, 12, 373-377.

Habermann, A.N. (1972), Synchronisation of communicating processes, *Comm. ACM*, 15, 171-176.

Hansen, P.B. (1970), The nucleus of a multiprogramming system, *Comm. ACM*, 13, 238-250.

Hansen, P.B. (1972), Structured multiprogramming, *Comm. ACM*, 15, 574-578.

Hansen, P.B. (1973), *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J.

Hansen, P.B. (1975), The programming language Concurrent Pascal, *IEEE Trans. Software Engineering*, 1, 199-207.

Hantler, S.L., and King, J.C. (1976), An introduction to proving the correctness of programs, *Computing Surveys*, 8, 331-353.

Hartley, D.F. (1970), Management software in multiple-access systems, *Bulletin of the I.M.A.*, 6, 11-13.

Havender, J.W. (1968), Avoiding deadlock in multitasking systems, *IBM Systems Journal*, 7, 74-84.

Hoare, C.A.R. (1972), Proof of correctness of data representation, *Acta Informatica*, 1, 271-281.

Hoare, C.A.R. (1974), Monitors: an operating system structuring concept, *Comm. ACM*, 17, 549-557.

Horning, J., Lauer, H.C., Melliar-Smith, P.M., and Randell, B. (1974), A program structure for error detection and recovery, in *Lecture Notes in Computer Science*, 16, Springer-Verlag, New York.

Huxtable, D.H.R., and Pinkerton, J.M.M. (1977), The hardware/software interface of the ICL 2900 range of computers, *Computer Journal*, 20, 290-295.

IBM Corporation, *IBM System 360/Operating System Concepts and Facilities*, Manual C28-6535.

ICL (1969), *Operating Systems George 3 and 4*, Manual 4169.

IFIP (1969), *File Organisation*, Selected papers from File 68, IFIP Administrative Data Processing Group, occasional publication number 3, Swets and Zeitliger.

Jackson, M. (1975), *Principles of Program Design*, Academic Press, London and New York.

Judd, D.R. (1973), *Use of Files*, MacDonald-Elsevier, London and New York.

Keedy, J.L. (1976), The management and technological approach to the design of System B, *Proc. 7th Australian Computer Conf.*, Perth, 997-1013.

Kernighan, B.W., and Ritchie, D.M. (1978), *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J.

Knuth, D.E. (1968), *The Art of Computer Programming*, vol. 1: Fundamental algorithms, Addison-Wesley, Reading, Mass.

Kopetz, H. (1979), *Software Reliability*, Macmillan, London and Basingstoke.

Lampson, B.W., and Redell, D. (1980), Experience with processes and monitors in Mesa, *Comm. ACM*, 23, 105-117.

Lister, A.M. (1974), Validation of systems of parallel processes, *Computer Journal*, 17, 148-151.

Martin, J. (1977), *Computer Data-Base Organisation*, 2nd ed., Prentice-Hall, Englewood Cliffs, N.J.

Martin, J. (1983), *Managing the Data-Base Environment*, Prentice-Hall, Englewood Cliffs, N.J.

Randell, B. (1975), System structure for software fault tolerance, *IEEE Trans. Software Engineering*, 1, 220-232.

Randell, B., Lee, P.A., and Treleaven, P.C. (1978), Reliability issues in computing system design, *Computing Surveys*, 10, 123-165.

Richards, M. (1969), BCPL: a tool for compiler writing and systems programming, *Proceedings AFIPS Spring Joint Computer Conference*, 34, 557-566.

Ritchie, D.M., and Thompson, K. (1974), The UNIX time-sharing system, *Comm. ACM*, 17, 365-375.

Schroeder, M.D., and Saltzer, J.H. (1972), A hardware architecture for implementing protection rings, *Comm. ACM*, 15, 157-170.

Stone, H.S. (ed.) (1975), *An Introduction to Computer Architecture*, SRA, Chicago.

Svobodova, L. (1976), *Computer Performance Measurement and Evaluation*, Elsevier, New York.

Tanenbaum, A.S. (1975), *Structured Computer Organisation*, Prentice-Hall, Englewood Cliffs, N.J.

Watson, R.W. (1970), *Timesharing System Design Concepts*, McGraw-Hill, New York.

Welsh, J., and Bustard, D.W. (1979), Pascal-plus - another language for modular multiprogramming, *Software Practice and Experience*, 9, 947-957.

Williams, R.K. (1972), System 250 - basic concepts, *Proceedings of the Conference on Computers - Systems and Technology*, I.E.R.E. Conference Proceedings number 25, 157-168.

Wirth, N. (1971), Program development by stepwise refinement, *Comm. ACM*, 14, 221-227.

Wirth, N. (1977), Modula: a language for modular multiprogramming, *Software Practice and Experience*, 7, 3-36.

Wirth, N. (1983), *Programming in Modula-2*, 2nd ed., Springer-Verlag, Heidelberg, Berlin and New York.

Wulf, W.A. (1975), Reliable hardware-software architecture, *IEEE Trans. Software Engineering*, 1, 233-240.

Wulf, W.A., Russell, D.B., and Habermann, A.N. (1971), BLISS: a language for systems programming, *Comm. ACM*, 14, 780-790.

Yourdon, E. (1975), *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, N.J.

Nederlandstalig

Alblas, H. (1983), *Systeemprogrammatuur*, 4e ed., Academic Service, Den Haag.

Ashley, R., en Fernandez, J.N. (1985), *PC DOS*, Academic Service, Den Haag.

Ashley, R., en Fernandez, J.N. (1985), *MS DOS*, Academic Service, Den Haag.

Austen, G.J.M., en Thomassen, H.J. (1985), *UNIX, het Standaard Operating Systeem*, Academic Service, Den Haag.

Clarke,A., Eaton, J.M., en Powys-Lybbe, D. (1985), *CP/M voor Gevorderden*, Academic Service, Den Haag.

EIT diktatenserie 4 (1980), *Bedrijfssystemen*, 3e ed., Academic Service, Den Haag.

Fernandez, J.N., en Ashley, R. (1984), *CP/M, het Operating System voor Microcomputers*, 4e ed., Academic Service, Den Haag.

Fernandez, J.N., en Ashley, R. (1985), *CP/M 86*, Academic Service, Den Haag.

Kernighan, B.W., en Pike, R. (in voorbereiding), *Werken met UNIX*, Academic Service, Den Haag.

Ledgard, H.F., Nagin, P.A., en Hueras, J.F. (1982), *Het Groot Pascal Spreuken Boek*, Academic Service, Den Haag.

Verhulst, E. (1984), *Systeemprogrammatuur en Software-ontwikkeling voor Microcomputers*, Academic Service, Den Haag.

Index

ACADEMIC SERVICE INFORMATICA UITGAVEN

AUTOMATISERING EN COMPUTERS

Computers en onze samenleving - M.A. Arbib
Computers in de negentiger jaren - G.L. Simons
De informatiemaatschappij - Jan Everink
Basiskennis informatieverwerking - Jan Everink
AIV, Automatisering van de informatieverzorging - Th.J.G. Derksen en H.W. Crins
Organisatie, informatie en computers - D.M. Kroenke
De Viewdata revolutie - S. Fedida en R. Malik

MICROCOMPUTERS

Microcomputers thuis en op school - K.P. Goldberg en R.D. Sherwood
Bouw zelf een Expertsysteem in BASIC - C. Naylor
Programmeercursus Microsoft BASIC - Nok van Veen
Werken met bestanden in BASIC - L. Finkel en J.R. Brown
40 Grafische programma's voor de Commodore 64 - M. Sutter
Doe-het-zelf programma's op de Commodore 64 - D. Kreutner
Programmeercursus BASIC op de Commodore 64 - Nok van Veen
TRS-80 BASIC - Bob Albrecht e.a.
TRS-80 BASIC voor gevorderden - Don Inman e.a.
Exidy sorcerer en BASIC - Nok van Veen e.a.
40 Grafische programma's voor de Electron en BBC - M. Sutter
Het Electron en BBC Micro boek - Jim McGregor en Alan Watt
Ontdek de ZX-Spectrum - Tim Hartnell
Werken met bestanden op de Apple - L. Finkel en J.R. Brown
Programmeercursus Applesoft BASIC - Nok van Veen
40 Grafische programma's voor de Apple II,IIe,IIc - M. Sutter
40 Grafische programma's in MSX BASIC - M. Sutter
Programmeercursus MSX BASIC - Nok van Veen

MICROPROCESSORS EN ASSEMBLEERTALEN

Procescomputers, basisbegrippen - dr.ir. J.E. Rooda en ir. W.C. Boot
Cursus Z-80 assembleertaal - Roger Hutty
6502 Assembleertaal en machinecode voor beginners - A.P. Stephenson

BESTURINGSSYSTEMEN

Inleiding besturingssystemen - A.M. Lister
Systeemprogrammatuur en software-ontwikkeling voor microcomputers - E. Verhulst
Bedrijfssystemen - EIT-serie, deel 4
CP/M het operating system voor microcomputers - J.N. Fernandez en R. Ashley
CP/M 86 - Nok van Veen
CP/M voor gevorderden - A. Clarke e.a.
PC DOS, het besturingssysteem van de IBM PC - R. Ashley en J.N. Fernandez
MS/DOS, het besturingssysteem voor 16 bit microcomputers - R. Ashley en J.N. Fernandez
UNIX, het standaard operating system - G.J.M. Austen en H.J. Thomassen
Werken met UNIX - Brian W. Kernighan en Rob Pike

PERSONAL COMPUTERS

Het werken met bestanden op de IBM PC - L. Finkel en J.R. Brown
De IBM PC en zijn toepassingen - Laurence Press
40 Grafische programma's voor de IBM PC - M. Sutter
Werken met VisiCalc - C. Klitzner en M.J. Plociak
Multiplan, een hulpmiddel bij de bedrijfsvoering - D.F. Cobb e.a.
Multiplan diskettes
Werken met Lotus 1-2-3 - D. Cobb en G. LeBlond

PROGRAMMEREN

Een methode van programmeren - prof.dr. Edsger W. Dijkstra en ir. W.H.J. Feijen
Programmeren, het ontwerpen van algoritmen (met Pascal) - ir. J.J. van Amstel
Inleiding tot het programmeren, deel 1 - ir. J.J. van Amstel
Inleiding tot het programmeren, deel 2 - ir. J.J. van Amstel
Programmeren, deel 2: van analyse tot algoritme - prof.drs. C. Bron
Inleiding programmeren en programmeertechnieken - EIT-serie, deel 1
Het Groot Pascal Spreuken Boek - H.F. Ledgard e.a.
JSP - Jackson Struktureel Programmeren - Henk Jansen
JSP Uitwerkingenboek - Henk Jansen

PROGRAMMEERTALEN

Aspecten van programmeertalen - ir. J.J. van Amstel en ir. J.A.A.M. Poirters
Programmeertalen, een inleiding - ir. J.J. van Amstel e.a.
BASIC - EIT-serie, deel 3
Cursus BASIC, een practicum-handleiding voor BASIC op de PRIME - ir. R. Bloothoofd e.a.
Cursus Pascal - prof.dr. A. van der Sluis en drs. C.A.C. Görts
Cursus eenvoudig Pascal - prof.dr. A. van der Sluis en drs. C.A.C. Görts
Inleiding programmeren in Pascal - C. van de Wijgaart
Systeemontwikkeling met Ada - Grady Booch
Cursus COBOL - A. Parkin
Cursus FORTRAN 77 - J.N.P. Hume en R.C. Holt
Aanvulling cursus FORTRAN 77 voor PRIME-computers - ing. J.M. den Haan
De programmeertaal C - ir. L. Ammeraal
Flitsend Forth - Alan Winfield
Programmeren in LISP - prof.dr. L.L. Steels

GEGEVENSSTRUCTUREN EN BESTANDSORGANISATIE

Informatiestructuren, bestandsorganisatie en bestandsontwerp - EIT-serie, deel 5
Programmeren, het ontwerpen van datastructuren en algoritmen - ir. J.J. van Amstel e.a.
Bestandsorganisatie - prof.dr. R.J. Lunbeck en drs. F. Remmen

DATABASE EN GEGEVENSANALYSE

Database, een inleiding - C.J. Date
Databases - drs. F. Remmen
Gegevensanalyse - R.P. Langerhorst

INFORMATIE-ANALYSE EN SYSTEEMONTWERP

Effectieve toepassingen van computers - M. Peltu
Voorbereiding van computertoepassingen - prof.dr. A.B. Frielink
Systeemontwikkeling volgens SDM - H.B. Eilers
Samenvatting SDM - Pandata
Informatie-analyse volgens NIAM - J.J.V.R. Wintraecken
Evaluation of methods and techniques for the analysis, design and implementation of information systems - ed. J. Blank en M.J. Krijger
Inleiding systeemanalyse, systeemontwerp - W.S. Davis
Systeemontwikkeling Zonder Zorgen - Paul T. Ward
Het ontwerpen van interactieve toepassingen en computernetwerken - J.A. Scheltens
EDP Audit - prof.dr. C. de Backer
Prototyping, een instrument voor systeemontwerpers - ed. T. Hoenderkamp en H.G. Sol
Simulatie, een moderne methode van onderzoek - drs. S.K. Boersma en ir. T. Hoenderkamp

EXPERT SYSTEMEN EN KUNSTMATIGE INTELLIGENTIE

Computerschaak - H.J. van den Herik
Expert systemen - Henk de Swaan Arons en Peter van Lith

THEORETISCHE INFORMATICA EN SYSTEEMPROGRAMMATUUR

Informatica, een theoretische inleiding - dr. L.P.J. Groenewegen en prof.dr. A. Ollongren
Systeemprogrammatuur - drs. H. Alblas
Vertalerbouw - H. Alblas e.a.

AANVERWANTE ONDERWERPEN EN OVERIGE TITELS

Lineaire programmering als hulpmiddel bij de besluitvorming - prof.dr. S.W. Douma
Inleiding programmeren - prof.dr. R.J. Lunbeck
Analyse van informatiebehoeften en de inhoudsbeschrijving van een databank - prof.dr. P.G. Bosch en ir. H.M. Heemskerk
Gegevensstructuren - R. Engmann e.a.
Cases en *Uitwerkingenboek bij Cases* - prof.dr. P.G. Bosch en H.A. te Rijdt
De tekstmachine - dr. M. Boot en drs. H. Koppelaar
Abstracte automaten en grammatica's - prof.dr. A. Ollongren en ir. Th.P. van der Weide
Onderneming en overheid in systeem-dynamisch perspectief - red. A.F.G. Hanken e.a.
Simulatie en sociale systemen - red. J.L.A. Geurts en J.H.L. Oud
Struktuur en stijl in COBOL - ir. E. Dürr en dr.ir. F. Mulder
Cursus ALGOL 60 - prof.dr. A. van der Sluis en drs. C.A.C. Görts

INFORMATIE OVER DEZE PUBLIKATIES BIJ:

Academic Service, Postbus 96996, 2509 JJ Den Haag, tel. 070-247238

NEDERLANDS	ENGELS
Geheugenbescherming	Memory protection
geheugenruimte	memory space
geheugentoewijzing	memory allocation
geheugenversnippering	memory fragmentation
gelijktijdigheid	concurrency
grensregister	limit register
- hardware-storing	fault hardware
herkennen van vastlopen	deadlock detection
herstel	recovery
herstelblok	recovery block
herstellen van vastlopen	deadlock recovery
herstelpunt	recovery point
huidige proces	current process
hulpbron-besturing	resource control
hulpbron-toewijzing	resource allocation
- I/O buffering	I/O buffering
I/O fout	error I/O
I/O procedure	I/O procedure
I/O toestand	I/O mode
I/O verzoekblok	I/O request block
indelen in pagina's	paging
indexblok	index block
interruptmechanisme	interrupt mechanism
interruptprioriteit	interrupt priority
interruptroutine	interrupt routine
interrupt, 'ingreepsignaal'	interrupt
invoer op afstand	remote job entry
invoer/uitvoer	I/O
- kanaal	channel
kern	nucleus
korrektheid	correctness
kritische sector	critical sector
- macro	macro (in job description)
maken van werkindeling	scheduling
maskeren van ---	masking of ---
meervoudig toegankelijk systeem	multi-access system
multiprogrammering	multiprogramming
multi-stream monitor	multi-stream monitor
- nulproces	null process
- onafhankelijkheid van I/O apparatuur	I/O device independance
ondeelbare hulpbronnen	resource non shareable
onschendbaarheid, bestand	integrity, file
ontsluithandeling	unlock operation
opdrachtentaal	command language
ophaalbeleid (geheugen)	memory fetch policies
opnieuw binnenkomen	re-entrant procedure
opslagapparaat	file device
opslag op één niveau	one level store
opslagsysteem	filing system
opzichterstoestand	supervisor mode
overlappen	overlay
overtollig, overbodig	redundancy
- pagina	page
pagina adres-register	page address register
paginafout	page fault
paginakader	page frame
pagina omzet-algoritme	page turning algorithm
paginatabel	page table
paginaverdeling	address paging

ENGELS	NEDERLANDS
File allocation	Toewijzing van een bestand
file deletion	wissen van een bestand
file descriptor	bestandsbeschrijver
file device	opslagapparaat
file directory	bestandsindex
file interrogation system	systeem voor raadplegen van gegevens
file map	bestandstabel
file name	bestandsnaam
filing system	opslagsysteem
first level interrupt handler	basis-niveau ingreep besturing
- gate	poort
- hole	gat
hole list	gatenlijst
- I/O	invoer/uitvoer
I/O buffering	I/O buffering
I/O device independance	onafhankelijkheid van I/O apparatuur
I/O mode	I/O toestand
I/O procedure	I/O procedure
I/O request block	I/O verzoekblok
incremental dump	gedeeltelijke dump
index block	indexblok
integrity, file	onschendbaarheid, bestand
interrupt	interrupt, 'ingreepsignaal'
interrupt mechanism	interruptmechanisme
interrupt priority	interruptprioriteit
interrupt routine	interruptroutine
- job control	besturen van opdrachten
job control	taakbesturing
job control language	besturingstaal voor opdrachten
job description	taakbeschrijving
job pool	takenpot
- limit register	grensregister
lock operation	afsluithandeling
- macro (in job description)	macro
masking of ---	maskeren van ---
massive dump	algehele dump
memory allocation	geheugentoewijzing
memory compaction	verdichting van geheugen
memory fetch policies	ophaalbeleid (geheugen)
memory fragmentation	geheugenversnippering
memory management	geheugenbeheer
memory placement policy	plaatsingsbeleid (geheugen)
memory protection	geheugenbescherming
memory relocation	verplaatsen van geheugen
memory space	geheugenruimte
monitor	bewakingsprogramma, monitor
multiprogramming	multiprogrammering
multi-access system	meervoudig toegankelijk systeem
multi-stream monitor	multi-stream monitor
mutual exclusion	wederzijdse uitsluiting
- nondeterminacy	flexibele respons
nucleus	kern
null proces	nulproces
- one level store	opslag op één niveau
overlay	overlappen
- P-operation	passeer-handeling
page	pagina
page address register	pagina adres-register
page fault	paginafout

Woordenlijst Engels-Nederlands bij:
Inleiding besturingssystemen - A.M. Lister

ENGELS	NEDERLANDS
Acceptance	Acceptatie
access matrix	toegangsmatrix
address map	adresvertaler
address paging	paginaverdeling
address space	adresruimte, namespace
allocation	toewijzing
associative store	associatief geheugen
audit trail	verificatiespoor
- backing store	achtergrondgeheugen
base register	basisregister
bit map	bit-tabel
busy waiting	actief wachten
- call bracket	aanroepbereik
capability	functie
capability list	functielijst
capability register	functieregister
capability segment	functiesegment
central file descriptor	centrale bestandsbeschrijver
central processor	centrale verwerkingseenheid
channel	kanaal
command language	opdrachtentaal
concurrency	gelijktijdigheid
control card	besturingskaart
correctness	correctheid
CP/M	CP/M
critical section	kritische sector
CTSS	CTSS
current process	huidige proces
- data base	gegevensbestand
data set	dataset
deadlock	vastlopen van het systeem
deadlock avoidance	vermijden van vastlopen
deadlock detection	herkennen van vastlopen
deadlock prevention	voorkómen van vastlopen
deadlock recovery	herstellen van vastlopen
device	apparaat
device handler	apparatuur besturingsroutine
dispatcher	verdeelprogramma
distributed system	gedecentraliseerd systeem
domain of protection	beschermingsdomein
double buffering	dubbele buffering
- efficiency	efficiëntie
error	fout
error detection	foutherkenning
error I/O	I/O fout
error trap	foutval
extracode	extracode, opzichtersverzoek
- fault	storing
fault avoidance	vermijden van storingen
fault hardware	hardware-storing
fault software	software-storing
fault treatment	behandelen van storing
fault user	bedieningsstoring
feedback	feedback, terugkoppeling
file	bestand

NEDERLANDS	ENGELS
Passeer (Dijkstra)	Wait
passeerhandeling	P-operation
periodieke dump	periodic dump
plaatsingsbeleid (geheugen)	memory placement policy
poort	gate
procestoestand, status	process status
procesbeschrijver	process descriptor
processor-wachtrij	processor queue
processtructuur	process structure
programma	program
programma-adres	program address
- responstijd	response time
- segment	segment
segmentatie	segmentation
segmentbescherming	segment protection
segmentbeschrijver	segment descriptor
segmenttabel	segment table
seinpaal-wachtrij	semaphore-queue
seinpaal	semaphore
skipketting	skip chain
software-storing	fault software
spooling	spooling
storing	fault
stroombeschrijver	stream descriptor
synchronisatie	synchronisation
systeem voor raadplegen van gegevens	file interrogation system
systeem voor procesbesturing	process control system
- taakbeschrijving	job description
taakbesturing	job control
takenpot	job pool
toegangsmatrix	access matrix
toestandstekening	state graph
toewijzing	allocation
toewijzing van een bestand	file allocation
- uitvoertijd	turn-around time
- vastlopen van het systeem	deadlock
verdeelprogramma	dispatcher
verdichting van geheugen	memory compaction
verificatiespoor	audit trail
vermijden van storingen	fault avoidance
vermijden van vastlopen	deadlock avoidance
verplaatsen van geheugen	memory relocation
verschrotten	thrashing
verhoog (Dijkstra)	signal
verhooghandeling	V-operation
verwerkingseenheid, processor	processor
virtueel geheugen	virtual memory
vluchtige omgeving	volatile environment
voorkómen van vastlopen	deadlock prevention
- wederzijdse uitsluiting	mutual exclusion
werkset	working set
werkverdeler	scheduler
wissen van een bestand	file deletion

ENGELS	NEDERLANDS
Page frame	Paginakader
page table	paginatabel
page turning algorithm	pagina omzet-algoritme
paging	indelen in pagina's
periodic dump	periodieke dump
privileged instruction	bevoorrechte instructie
process control system	systeem voor procesbesturing
process descriptor	procesbeschrijver
process status	procestoestand, status
process structure	processtructuur
processor	verwerkingseenheid , processor
processor queue	processor-wachtrij
program	programma
program address	programma-adres
protection	bescherming
- re-entrant procedure	opnieuw binnenkomen
recovery	herstel
recovery block	herstelblok
recovery point	herstelpunt
redundancy	overtollig, overbodig
reliability	betrouwbaarheid
remote job entry	invoer op afstand
resource allocation	hulpbron-toewijzing
resource control	hulpbron-besturing
resource non shareable	ondeelbare hulpbronnen
resource shareable	deelbare hulpbronnen
response time	responstijd
ring of protection	beschermingsring
- scheduler	werkindeler
scheduling	maken van werkindeling
segment	segment
segment descriptor	segmentbeschrijver
segment protection	segmentbescherming
segment table	segmenttabel
segmentation	segmentatie
semaphore	seinpaal
semaphore queue	seinpaal-wachtrij
signal	verhoog (Dijkstra)
skip chain	skipketting
spooling	spooling
state graph	toestandstekening
stream descriptor	stroombeschrijver
supervisor mode	opzichterstoestand
synchronisation	synchronisatie
- thrashing	verschrotten
turn-around time	uitvoertijd
- unlock operation	ontsluithandeling
user mode	gebruikerstoestand
V-operation	verhooghandeling
virtual memory	virtueel geheugen
volatile environment	vluchtige omgeving
- wait	passeer (Dijkstra)
working set	werkset

Woordenlijst Nederlands-Engels bij:
Inleiding besturingssystemen - A.M. Lister

NEDERLANDS	ENGELS
Aanroepbereik	Call bracket
acceptatie	acceptance
achtergrondgeheugen	backing store
actief wachten	busy waiting
adresruimte, namespace	address space
adresvertaler	address map
afsluithandeling	lock operation
algehele dump	massive dump
apparaat	device
apparatuur besturingsroutine	device handler
associatief geheugen	associative store
- basis-niveau ingreep besturing	first level interrupt handler
basisregister	base register
bedieningsstoring	fault user
behandelen van storingen	fault treatment
bescherming	protection
beschermingsdomein	domain of protection
beschermingsring	ring of protection
bestand	file
bestandsindex	file directory
bestandsbeschrijver	file descriptor
bestandsnaam	file name
bestandstabel	file map
besturen van opdrachten	job control
besturingskaart	control card
besturingstaal voor opdrachten	job control language
betrouwbaarheid	reliability
bevoorrechte instructie	privileged instruction
bewakingsprogramma, monitor	monitor
bit-tabel	bit map
- centrale bestandsbeschrijver	central file descriptor
centrale verwerkingseenheid	central processor
CP/M	CP/M
CTSS	CTSS
- dataset	data set
deelbare hulpbronnen	resource shareable
dubbele buffering	double buffering
- efficiëntie	efficiency
extracode, opzichtersverzoek	extracode
- feedback, terugkoppeling	feedback
flexibele respons	nondeterminacy
fout	error
foutval	error trap
foutherkenning	error detection
functie	capability
functielijst	capability list
functieregister	capability register
functiesegment	capability segment
- gat	hole
gatenlijst	hole list
gebruikerstoestand	user mode
gedecentraliseerd systeem	distributed system
gedeeltelijke dump	incremental dump
gegevensbestand	data base
geheugenbeheer	memory management

OVER HET BOEK:

Dit boek is een vertaling van het bekende boek 'Fundamentals of Operating Systems' van A.M. Lister. Het boek begint met het definiëren van de functies, die gebruikers van een besturingssysteem (operating system) van het systeem verwachten. Een aantal overeenkomstige eigenschappen van besturingssystemen komt hierbij aan de orde.
Vervolgens wordt het begrip 'proces' geïntroduceerd als basis voor het gelijktijdig door de computer kunnen uitvoeren van een aantal activiteiten (concurrent processes).
Verder beschrijft het boek een hypothetisch besturingssysteem, vanuit de koppeling (interface) met de hardware tot en met de koppeling met de gebruiker. Hierbij komen aan de orde: processorbeheer, geheugenbeheer, in- en uitvoer, bestandsregistratie, apparatuurtoewijzing, beveiliging en taakbesturing.
Steeds ligt de nadruk op de logische, hiërarchische structuur van besturingssystemen, hetgeen ook volgens de auteur de beste manier is om betrouwbare en hanteerbare besturingssystemen te bouwen.

OVER DE AUTEUR:

A.M. Lister heeft voorheen colleges gegeven op de universiteiten van Lancaster en Essex. Vanaf 1976 geeft hij colleges op de universiteit van Queensland, waar hij nu hoofd is van de afdeling Informatica. Hij publiceert in zo'n dertig tijdschriften en is mede-auteur van een ander inleidend leerboek.

ISBN 90 6233 165 3